An Aspect Refactoring Tool for The Observer Pattern

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of MSc
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Fatima Alawami

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Current integrated development environments such as Eclipse provide strong support for object-oriented automatic refactorings; however, the same cannot be said about aspect-oriented refactorings. Refactoring of design patterns is one area where aspect refactoring automation remains to be explored in depth and few current tools are available to support it. To support aspect refactoring tools we present the AJRefactor plug-in, a semi-automatic refactoring tool for the observer pattern, a widely-used solution in the design of object-oriented programs. Aspect refactoring of the observer pattern allows aspects to capture pattern-specific code into a more modularized unit, and localizes the code of participating classes. After applying AJRefactor on two Java projects JHotDraw and Prevayler, the results showed that AJRefactor was able to refactor 75% of the total observer instances found in both projects. Also, the refactoring enhanced the modularity and loosens the coupling of the pattern classes. Finally, the results showed a significant time savings and a small reduction in code size when refactoring with AJRefactor.

Acknowledgements

I would like to use this opportunity to give special thanks to several people who contributed to the completion of this work.

- I am heartily thankful to my supervisor, Christopher Dutchyn , whose encouragement, guidance and support from the initial to the final level enabled me to develop a deep understanding of the subject of my thesis. I really appreciate your enlightening advice through the entire period of my master's studies. Working with my supervisor has improved my writing, thinking and learning skills.

- I would like to sincerely thank the Ministry of High Education of Saudi Arabia for supporting this work financially.

- I wish to avail myself of this opportunity, to express a sense of gratitude and love to my husband and my beloved parents for their continuous support.

- Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of my thesis.

Contents

List of Tables

List of Figures

List of Abbreviations

AST	Abstract Syntax Tree page 14
ITD	Inter-Type Declaration page 10
CC	Crosscutting Concerns page 20
GUI	Graphical User Interface page 32
SLOC/LOC	Source Lines of Code page 74
API	Application Programming Interface page 21
DSM	Dependency Structure Matrix page 76
AspectBrowser	Set of tools to help developers focus on software modification task page 20
ConcernMapper	Maps program elements into concern constructs created by the user page 20
FEAT	Feature Exploration and Analysis Tool page 21
The Prism Aspect Miner	A generative based aspect mining tool that mine aspects using a probabilistic approach by extending the page ranking algorithm page 22
Fan-in Analysis Tool	Mines for aspects by calculating methods fan-in metric and then filtering the results until those that are more likely to be part of a concern ramains page 22
Dynamo	Mines for crosscutting concerns based in the formal concept analysis, a branch of lattice theory. page 23
AJaTS	An AspectJ transformation and code generation tool page 23
AspectRefactor	A refactoring tool that automatically refactors the Java authentication and authorization part of the code page 24
CRAFT	An open infrastructure to automatically refactor Java programs into aspects page 24
JastAddJRefactoring	A refactoring tool that make use of the analysis features of the JastAddJ compiler page 25
The Aspect-orinted Migrator Tool	A refactoring tool with six refactoring actions to migrate programs from Java to aspects page 25

Chapter 1

Introduction

In software engineering, we try to keep the code as simple as possible. System complexity is reduced by breaking it down into smaller modules (e.g. methods and procedures) each focusing on one single task. This is often combined with performing a series of source code transformations with the intent of improving the internal design of the system while maintaining its external behaviour, a process known as code refactoring. Improving system structure helps achieving more modular code that is easier to comprehend, read and maintain.

Developers are recommended to perform a refactoring in a step-by-step fashion along with a group of unit tests for each step to ensure the system still works correctly. For example, when renaming a method to meaningful name that reflects its functionality, the developer should

(1) create a new method with an empty body.

(2) copy the old method's body into the newly created method and test.

(3) replace the old method's body with a call to the new one and test.

(4) replace all references to the old method with the new one and test and finally.

(5) given that all tests have passed, it is now safe to remove the old method.

Even for a simple refactoring (e.g. renaming a method), manual refactoring is time consuming and error prone, which raises the need for a safe and reliable tool that automates the refactoring process. Tools not only save development time and effort but also improve productivity. For that reason many integrated development environments such as Eclipse provide developers with a set of refactoring tools.

A key benefit of refactoring is modularity. Modularization keeps program constructs focused around one specific task, this increases their reusability. In the context of object-oriented programming we find two types of behaviours and concerns: core concerns and crosscutting concerns. Core concerns capture the main functionality of a module (e.g. customer and account management in a bank system) while crosscutting ones capture system-level functionality (e.g. authentication). To authenticate the person performing any kind of transactions in a bank system, the developer has to call the authentication module before every transaction. Although this is one of the cleanest to

implement the authentication feature, it can still benefit from extra modularity. Aspect-oriented programming constructs provide a better opportunity to modularize crosscutting concerns by capturing each concern into its own module and then loosely coupling them to a limited number of other modules.

In the context of object-oriented programs, the developer often faces some common problems for which an effective solution is unlikely to be found. Many of these problems are solved using design patterns [17, p. 2]. Java implementations of the design patterns presents some issues, including fragmentations, invasiveness and obscurity. Adding a pattern tends to spread over many classes making such change difficult to revert. Although the design patterns are reusable, their implementations are not. Often the pattern implementation is invasive, tracking the pattern instances is a difficult task, and leads to obstacles in documentation. This is especially true if the class is involved in more than one pattern.

According to Hannemann and Kiczales [20], refactoring Java systems that make a good use of design patterns into AspectJ, showed that 17 of 23 design patterns gained better modularity, and 12 allowed the core part of the pattern implementation to be abstracted into a reusable module. In addition, localizing the pattern participant code and freeing it from any pattern-related code has two advantages. First, the participant can be transparently composed in multiple patterns without the need to change its code. Second, it is easy to add or remove the participant from the pattern at any moment. Complex patterns, where

- a single object plays multiple roles, or
- multiple objects play the same role,

show the most improvement [20].

1.1 Thesis Statement

Aspect refactoring of the observer pattern allows

- aspects to capture pattern-specific code into a more modularized unit, and
- localizes the code of participating classes.

1.2 Outline

The rest of the thesis is as follows: chapter two discusses background and similar research done on the area of aspect refactoring; chapter three explains our tool in depth in terms of its functionality and implementation; chapter four explains the tool design; chapter five discusses the results of the case studies and chapter six summarizes this thesis and outlines our future work.

Chapter 2

Background

Object-oriented programs work well for modelling common behaviours; but, they fall short when dealing with certain kind of behaviour that spans multiple, often unrelated, modules, known as crosscutting behaviours or *crosscutting concerns*. One of the areas that the Java programming platform does not support an effective modularity for is the implementation of many of the design patterns as they span multiple classes making the implementation fade into the participating classes. Having multiple classes carry part of the pattern code makes those classes tightly-coupled. This makes these classes hard to maintain or even to change; but, also makes documenting such system a challenging task. Additionally, when the same set of classes take part in multiple patterns, that renders the system harder to understand. Hannemann et. al.[20] observed introducing aspects not only enhances its implementation but also eases the documentation and composition of the classes.

Our work involves showing the effectiveness of AspectJ constructs in enhancing the modularity of Java programs. Therefore, the first part of this chapter illustrates with code examples the use of each one of these constructs. We use aspects to particularly reveal how we can completely move the observer pattern implementation into aspects. The second part of this chapter discusses the structure of the observer pattern along with its two common implementations. Then, we move into explaining the different tools and features provided by the Eclipse platform that help us in implementing our tool (AJRefactor). Finally, we review related work in the area of aspect refactoring tools and how it differs from our own work.

2.1 AspectJ

The discussion of AspectJ and its constructs is driven from *AspectJ in Action*[23]. AspectJ, an extension of Java, provides a new opportunity for breaking Java programs down into modules. AspectJ introduces new constructs that can improve the refactoring of Java programs. These constructs are of two types: dynamic and static [23]. Dynamic constructs are *aspect*, *pointcut*, and *advice*, while the static construct is the *inter-type declaration*. We use code listings in 2.1, 2.2 and 2.3 to illustrate the use and function of each of these constructs.

The `Point` class in listing 2.1 represents a point in two-dimensional space. It has the methods

```
class Point {
    private int x, y;

    Point(int x, int y) {
    this.x = x;
    this.y = y; }

    void setX(int x) { this.x = x;  }
    void setY(int y) { this.y = y; }

    int getX() { return x; }
    int getY() { return y; }
    }
```

Listing 2.1: Point Class

```
public class Screen {
    ...
    public void erase(){ ... }

    public void redraw() { ... }
}
```

Listing 2.2: Screen Class

to set and get the values of the (x) and (y) co-ordinates respectively. Listing 2.2 shows the `Screen` class that display these points. The screen needs to update itself if the point moves. Before the screen can redisplay itself, it has first to erase its previous contents and then redisplay the new content. Using this scenario, we explain how this behaviour is captured with aspects.

2.1.1 Pointcut and Join Points

The first aspect construct, pointcuts select regions in code that correspond to identifiable points called join points in the execution of the program. Pointcuts are used primarily to determine where new behaviour, called advice, should be activated. They can be composed using boolean operators to build other pointcuts. Below, we explain several join points that AspectJ exposes to weave in this new, crosscutting behaviour along with the pointcut constructs used to capture each of them.

Join Points

- *field read and write access:* this join point identifies the write and read access to non-static fields of a class or an aspect. This does not include setting or referencing a method local variable. An example of a reference of (x) member of the `Point` class is in the body of the `getX()` method (i.e. `return x;`). The statement `(this.x = x)` represents a write access to the x field of the `Point` class. The constructs `set(FieldPattern)` and `get(FieldPattern)` are used to capture the write and read access join points respectably.

- *method call:* this join point identifies the places in code where a method is called. For example, calling the method setX with an int argument from any other method (point.setX(5)).

- *method execution:* the scope of this join point is the whole body of a method. For example. the body of the `setX` method, (`{this.x = x;}`) is the scope of the join point matched by an `execution(void setX(int))` pointcut. This can be captured with the `execution(MethodPattern)` construct where
  `MethodPattern` represents the signature of the method we want to intercept.

  Previous examples show how method execution and method call pointcuts can be used to match the execution and call to instance methods. They can also be used to capture executions and calls of statically declared methods as well. For example, `call (static void *.foo())` matches all static methods named `foo()`.

- *constructor call:* this join point is exposed when an object is instantiated. That is when calling the *new* operation on an object. For example,
  `Point point = new Point(x, y);`
  This can be captured with the `call(ConstructorPattern)` construct. `ConstructorPattern` represents the signature of the constructor we want to capture.

- *constructor execution:* similar to the method execution join point, this join point exposes the body of the constructor of an object. This can be captured with the
  `execution(ConstructorPattern)`
  construct. `ConstructorPattern` represents the signature of constructor we want to capture. For example, `execution(Point.new(int,int))`.

- *class initialization:* this join point exposes the static initialization of an object. This can be captured with the `staticinitialization(TypePattern)` construct.

- *exception handler execution:* this join point exposes the handler block of an exception type (i.e. the catch block). This can be captured with the `handler(TypePattern)`.

- *advice execution:* this join point exposes the execution of the body of an advice. The construct `adviceexecution()` is used to capture the advice execution join point.

Table 2.1 maps exposed join points to pointcut designators.

Various pointcuts build on textual patterns in the program code. `FieldPattern`, `MethodPattern`, `TypePattern`, and `ConstructorPattern` are used to specify the signature of a field, a method, a constructor, or a type to be selected. The pattern can have wildcard characters such as `*` which means any. For example, the statement `call(public * *(int))` selects any method that takes an int argument regardless of its name and its return value.

Table 2.1: Mapping of exposed join points to pointcut construct

Join Point Type	Pointcut Syntax
Method call	`call(MethodPattern)`
Method execution	`execution(MethodPattern)`
Constructor call	`call(constructorPattern)`
Constructor execution	`execution(ConstructorPattern)`
Field read access	`get(FieldPattern)`
Field write access	`set(FieldPattern)`
Object initialization	`initialization(ConstructorPattern)`
Object pre-initialization	`preinitialization(ConstructorPattern)`
Class initialization	`staticInitialization(TypePattern)`
Exception handler execution	`handler(TypePattern)`
Advice execution	`adviceexecution()`

Pointcuts

Pointcuts identify or name join points. Here we explain the usage of the different pointcuts. Pointcuts are divided into six categories.

Control-flow Based Pointcuts This kind of pointcut captures join points that are defined in the control flow of other pointcuts. Control-flow pointcuts take a pointcut as an argument and comes in two variations: `cflow(pointcut)` and `cflowbelow(pointcut)`. The `cflow(pointcut)` pointcut matches all join points matched by the defined pointcut and all the join points in their control flow. For example, `cflow(call(* Point.setX(int)))` matches all join points in the control flow of any call to `setX(int)` in the `Point` class including the call to `setX(int)` method itself. These join points will be encountered in the following order:

- Method call of `setX(int)`
- Method execution of `setX(int)`
- Field set i.e. `this.x=x`

The `cflowbelow(pointcut)` pointcut in the other hand matches all join points in the control flow of those join points matched by the defined pointcut excluding those ones matched by the supplied pointcut. In the example, a `cflowbelow(call(* Point.setX(int)))` will match the same join points matched by `cflow` excluding the call to `setX(int)` method itself.

Lexical-structure Based Pointcuts The scope of these pointcuts is a block of source code of a class, aspect, or a method. This kind of pointcut captures join points of the code as it is written when

matched against the scope of the code during the execution, that is the dynamic scope. There are two types of lexical pointcuts `within(TypePattern)` and `withincode()` which has two forms: `withincode(MethodPattern)` and `withincode(ConstructorPattern)`. Join points are picked in the block of code of any class or aspect who is signature matches `TypePattern` or the source code of a method body or constructor body who is signature matches `MethodPattern` and `ConstructorPattern` respectively.

Conditional Check Pointcuts This pointcut captures each join point where the conditional expression of an if statement evaluates to true. It takes the form `if(BooleanExpression)`. The boolean expression can be but not limited to a constant value, a methods call, an api method call, and a variable. The context of the boolean expression should be collected by other parts of the pointcut. This include any object, class field, or method argument. For example,
`if(point.getX()>5)`.
This join point picks any join point where (x) co-ordinate of a point is greater than five. The point object must be a context collected by other parts of the pointcut.

Argument Pointcuts This pointcut captures join points based on the type and position of the arguments of the join point. The argument pointcut is used to capture context and pass it to the advice. For constructor and method join points, the arguments are the method and constructor arguments. For exception handler join points the handled exception object is considered the argument. For a field-write access join points the new value of that field is considered an argument. This pointcut takes the form `args(argsName)` where `argName` is a user defined for the declared argument.

Executing Object Pointcuts This pointcut captures join points based on the types of objects at the execution time. It takes two forms: `this(Type)` captures join points where the currently executing object is Type while the from `target(Type)` captures join points where the target is an instance of Type. The target pointcut is usually used with method call pointcut and the target object is the receiver of the method invocation.

Table 2.2 shows an example of these pointcuts along with a description of their scope.

Pointcut Composition Using logical operators we can compose different pointcuts to build additional pointcuts.

- to capture each join point not exposed within a pointcut we use `!pointcut`.
- to capture the join points exposed by number of pointcuts we use the && operator. For example, p1 && p2.

- to capture the join points exposed by any of the pointcuts we use the || operator. For example, p1 || p2.

The example aspect in listing 2.3 declares a pointcut called `subjectChange`. The statement `call(void setX(int))` defines that the pointcut selects the call to a method that return no value, take an int argument and has setX identifier. Similarly, `call(void setY(int))` means the pointcut selects a call to a method with an identifier setY with an int argument and return no value. Now, either one of them can be selected because of the or operation but the calling object should be of type `Point`. This is what the target construct is used for.

Each pointcut identifies a collection of join points, each is a place where an action called an advice can take effect.

2.1.2 Advice

Advice is the code executed when any of the join points identified by the pointcut is reached. This action can be executed before, after or around the execution of the selected join point.

- `before` advice is executed prior to the execution of the join point.

- `after` advice is executed after the execution of the join point.

- `around` advice surrounds the join point. It can replace the execution of the selected join point, bypass, or continue it. An empty body of an around advice means there is no action to be executed when the join point is matched. Using the `proceed()` construct allows the execution of the joint point to take place, possibly more than once. The signature of the proceed construct matches the signature of the parameter list of the around advice specification. The around advice can change the context in which the join point can proceed with by altering the value of the arguments passed to `proceed()` call. Since an around advice requires a return type which should match the type retuned by the matched join point, around advice can also alter this returned value.

The around advice in listing 2.3 shows the actions taken by `screen` object whenever a point moves (i.e. sets the value of its co-ordinates). The `screen` needs to update itself accordingly. It first erases itself and then redraw the points in their new positions. This scenario can be easily captured in the body of the around advice. Before the execution of the `setX` or `setY` methods, the `screen` erases itself. The `proceed()` allows the execution of `setX` and `setY` to take place. After their execution the `screen` can now redraw itself.

Table 2.2: Example pointcuts with Description

Pointcut Example	Description
`within(Point)`	any join point inside the `Point` class's lexical scope.
`withincode(Point.setX(int))`	any join point inside the lexical scope of the method `setX(int)` in the `Point` class.
`this(Point)`	all join points where `this` is instance of `Point`. This construct matches all join points where current executing object is `Point` such as method calls.
`target(Point)`	all join points where the receiver of the method call is an instance of class `Point`. This matches all join points where the target object is a `Point` or any of its subclasses.
`args(name, .. , age)`	this construct matches all methods where the first argument is of type `String` and the last argument is of type `int`.
`args(RemoteException)`	all join points that takes a single argument of type `RemoteException`. It would match a method taking a single `RemoteException` argument, a field being set with a value of type `RemoteException` or an exception handler triggered by `RemoteException`.
`if(x>0)`	matches all conditional tests where (x) greater than zero evaluates to true.
`cflow(subjectChange())`	all join points in the control flow of the join points captured by `subjectChange()` pointcut.
`cflowbelow(subjectChange())`	all join points in the control flow of the join points capture by `subjectChange()` excluding the call to `setX(int)` itself.

```
1  public aspect CoordinateObserver extends ObserverProtocol {
2
3      protected pointcut subjectChange(Subject s):
4          (call(void Point.setX(int))   ||
5          call(void Point.setY(int))    &&
6          target(s);
7
8      void around(Subject s): subjectChange(s){
9          Screen.erase();
10         proceed();
11         Screen.update();
12     }
13 }
```

Listing 2.3: CoordinateObserver Aspect

2.1.3 Inter-type Declaration

Recall that AspectJ supports static aspects also. Inter-type declarations (ITD) allow the developer to alter the static structure of the program or alter inheritance hierarchy. For example, ITDs introduce new class members (e.g. fields and methods). For example, aspect `ObservableOfChangeSomething` in figure 3.7 on page 47 declares a new method `update(Observable, Object)` to the `MyView`. The same aspect declares `MyView` to implement the `Observer` from Java utility package `java.util.Observer`.

2.1.4 Aspect

An `aspect` is the main unit of modularity for aspect-oriented programs. It behaves much like a class and can have fields and methods as well as pointcuts and advice. When compiling a Java program being extended by aspects, the aspects are woven into the Java program as if they were part of it. Similar to classes, aspects can be part of an aspect hierarchy. They can be declared as `abstract` and then extended by other aspects. Aspects can also declare abstract methods and abstract pointcuts which must be concretely specified in sub-aspects.

2.2 Observer Pattern

To demonstrate and evaluate how aspect refactoring for the design patterns achieves the features mentioned in chapter 1, we have chosen to implement an aspect refactoring tool for the observer design pattern. This pattern is one of the behavioural design patterns that defines one-to-many relationship between pattern participants [17, pp. 293-299]. Its main components are

- the observer class
- the subject class

- the subject interface, that the subject implement to be able to add, remove and notify its observers when a particular change to its state has occurred;
- the observer interface, which the observer class implements to update its state after being notified by any of the subject classes that it observes.

Although this is the most general form an observer pattern could take, it is not the only possible one. The implementation of the pattern can vary dramatically; there are two ways to trigger the update, and it is possible not to have the subject and observer interfaces. Two notable methodologies to trigger the update pushing or pulling. In the pushing technique the subject is the entity responsible for notifying its observers of state change while in the pulling technique observers have to query the subject state and then update its own state accordingly.

Hannemann et. al. [20] showed the kinds of improvements possible with AspectJ when refactoring the observer pattern to aspects. These improvements result in code that is:

- *localized*: every subject and observer is free of any pattern-related code.
- *reusable*: the functionality of the pattern is encapsulated into an abstract aspect while a particular instance of the pattern is easily implemented with a custom Aspect.
- *transparently composed*: the same class can play observer and subject role at the same time with its code remaining intact. It also can take part in other patterns easily.
- *unpluggable*: It is easy to add or remove a class from any pattern without needing to change the class base code.

2.3 Eclipse Platform

Our refactoring tool will be constructed as a plug-in to the popular Eclipse integrated development environment. It has a set of tools needed to develop Java applications including Eclipse itself. These include source code editors, compilers and debuggers. Eclipse also supports developing applications in many other languages such as C, C++, PHP and COBOL. This support is provided through the use of the plug-in platform.

Currently, there is wide support for object-oriented refactoring tools in many integrated development environments including Eclipse. However, the same is not true for aspect-oriented refactorings. Much effort has been put into documenting aspect refactorings [28] and conducting refactoring case studies on large object-oriented projects to extract crosscutting concerns into aspects, [27] but little progress has been achieved on supporting tools that provide some level of automation as most of them are research prototype and now are defunct. Therefore, our goal is is to provide support for aspect refactoring tools by building a semi-automatic eclipse plugin (AJRefactor) so it can be publicly available.

```
1 ISelectionService selectionService = (ISelectionService) IServiceLocator.getSite().
      getService(ISelectionService.class);
```

Listing 2.4: The Selection Service

2.3.1 Eclipse Plug-ins

We have chosen to implement a plug-in for Eclipse platform (AJRefactor). A plug-in is a group of software components that provide additional features and functionalities to a larger framework [14]. Eclipse is built from number of subsystems that are essentially built as one or more plug-ins. Plug-ins connect to the Eclipse framework by connecting to extension points. An extension point is a contract constructed of XML markups and Java interfaces that allow other plug-ins to extend or customize its functionality by simply complying to the extension point rules defined in that contract (i.e. implementing the interfaces). Using Eclipse extension point and extension mechanisms with the help of Java development tools, one can create, develop, test, debug, build and deploy plug-ins that add a variety of abilities and features to Eclipse platform.

When an Eclipse plug-in loads, Eclipse parses the manifest file of the plugin. Manifest file describes how the plug-in extends the Eclipse platform, what extension points it extends and how it implements its functionality. When loading the plug-in, Eclipse searches for the information needed to display the plug-in in the user interface. The classes that implements the plug-in are loaded only when the plug-in needs to run.

Our tool contributes to Eclipse workbench, a set of editors and views. Eclipse groups those editors and views into perspectives. For example, the Java perspective consists of

- project view,
- Java editor, and
- outline view.

In the plug-in development perspective user has another set of editors and views. The workbench offers a number of services. One can get any of these services using `org.eclipse.ui.services.IServiceLocator`. Services facilitate retrieving information about the workbench components without the need to use `PlatformUI.getWorkbench()`. Listing 2.4 shows an example.

AJRefactor uses the selection service to track the selection within the Java editor.

Commands and Actions

Eclipse provides two different frameworks to contribute to the workbench: the command and the action frameworks. These allows the plug-in to contribute many different behaviours and actions

to the workbench (e.g. views, editors and menus). Both allow part of code to execute when the user clicks a tool bar icon, or a menu item, or a key combination.

Commands declare a semantic behaviour which can be associated with a particular handler. Handlers which implement the command behaviour are defined using the `org.eclipse.ui.handlers` extension point. Activating the handler requires less code as this get declared in the manifest file. To place a command in any of the workbench parts only one extension point has to be defined `org.eclipse.ui.menus`.

Actions in the other hand declares both the manifest part and the code to be executed when action activates. The selection event is passed to the action to change the enablement state of the action based on the current selection.

To provide the functionality intended for our AJRefactor tool, we conform to the Eclipse refactoring framework. It also make intensive use of Eclipse source code manipulation facilities. Here we give a brief description of each. To place an action in the different parts of the workbench, programmer needs to extends different extension points depending on which part he wants to contribute an action to. For example, programmer needs to extend `org.eclipse.ui.viewActions` extension point to contribute an editor action while she needs to extend `org.eclipse.ui.popupMenus` extension point to contribute a popup menu.

2.3.2 Java Refactoring Framework

The Eclipse refactoring framework has two plug-ins: the core plugin `org.eclipse.ltk.core.refactoring` and its user-interface counterpart `org.eclipse.ltk.ui.refactoring`. The core plugin provides an infrastructure to contribute refactorings to the refactoring history, the refactoring scripting facility, and to the Eclipse workbench. It also provides the facility to test the refactoring in a local workspace, internally perform the precondition checking, and create and validate the change object. The refactoring interface provides abstract wizard and user input pages implementations, shows error messages and change preview.

Most of the refactorings executed are performed interactively as the user initiates them; but, some could be executed at different points of time using the refactoring script. This is possible by creating an object of type `org.eclipse.ltk.core.refactoring.RefactoringDescriptor`, which collets specific data that is specific to every refactoring instance. This data includes

- a human- readable description of the refactoring instance.
- the time it took to execute the refactoring.
- a unique id that is composed of the name of the refactoring being executed (Refactor Observer) and the refactoring unique id. For example, `ca.usask.se.ajrefactor.refactorObserver`

This descriptor is used to integrate the refactoring instance into Eclipse's workspace global history. Now, that the descriptor is created, the refactoring script can launch the refactoring as if the refactoring was generated by the user.

2.3.3 Java Source Manipulation

Our AJRefactor plug-in needs to manipulate Java elements to recognize pattern components. This can be done by using a combination of Eclipse's Java model and the traditional abstract syntax tree (AST).

Java Model

Java model is a collection of classes that represents the different objects that are involved in initiating, modifying and building Java programs. The structure of a Java program is formulated from a set of elements. These elements are modelled by a number of classes defined in `org.eclipse.jdt.core` package. Programs structure, which is obtained from the project's class path, is hierarchal as some elements are children of other elements. Some of these elements include

- `IJavaModel` the root java element. its children are all Java projects.
- `IJavaProject` a child of `IJavaModel`. It represents a Java project in the workspace.
- `IPackageFragmentRoot` represents a folder, a JAR, or a ZIP file that is composed of a number of package fragments.
- `IPackageFragment` a package fragment is a child of `IPackageFragmentRoot` and is composed of a number of packages declarations.
- `IPackageDeclaration` a child of `ICompilationUnit` that represents a package declaration.
- `ICompilationUnit` represents java source code file.
- `IType` which represents a java type whether it is a top, local or anonymous type.
- `IMethod` and `IField` which represent class members methods and fields respectively.

Figure 2.1 shows the various elements of Java model.

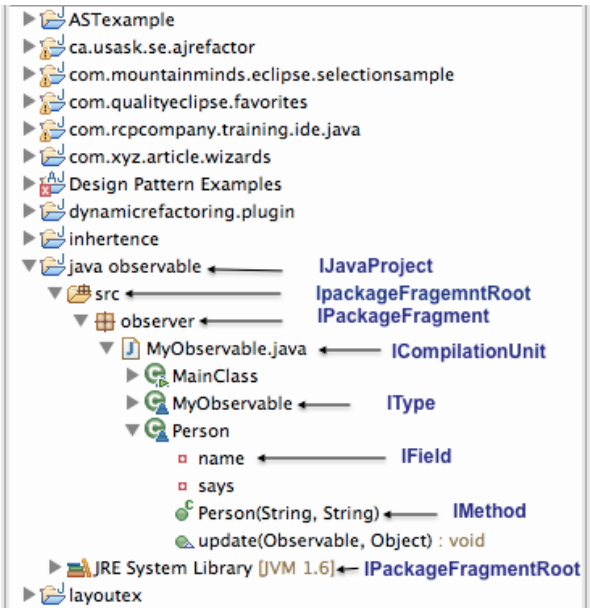

Figure 2.1: Java Model Overview

Abstract Syntax Tree

An abstract syntax tree is a tree representation of the source code produced primarily by the system-provided parser . We parse the code using the `org.eclipse.jdt.core.dom.ASTParser` provided by the Eclipse Java development tools. The application that uses the AST goes under the following steps [22]. Figure 2.2 gives a complete overview of the workflow of the AST manipulation.

(1) Obtain the concrete syntax as an array of characters from the source code or from the underlying `ICompilationUnit` or the binary type (.class file).

(2) Parse the code. The parser parses the given code into a tree structure reflecting the original text. It can take a block of code or a particular statement. When asking the parser to resolve type-binding, complete information about the different types of nodes is provided in the resulting AST. This includes the scoping of all the fields, methods and their formal parameters and local variables.

(3) Manipulate the AST by adding, deleting, and modifying the nodes in AST. The AST can be modified directly (4 A) or the changes can be recorded using the ASTRewrite mechanism (4 B).

(4) Write the modification back. Source code changes need to be applied to the underlying document that represents the source file. This is done by retrieving an object of type `TextEdit`

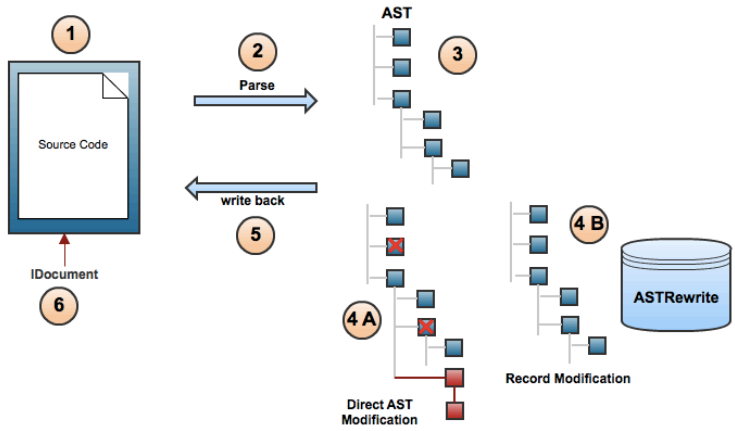

Figure 2.2: Workflow of AST Manipulation

either by asking the `ASTRewrite` to write the changes back or calling rewrite on the compilationUnit (AST resulting from the parsing step)

AST Nodes In this section we give a brief description of some of the tree nodes as we need them in the next chapters. This is by no means an extensive list of AST nodes. A complete description of all nodes can be found in Eclipse documentation [14].

Eclipse type `org.eclipse.jdt.core.dom.ASTNode` is the abstract superclass of all the AST node types. This type provides each node with a number of operations. Some of these operations include getting a node AST, accepting a visitor, deleting a node from its parent node and getting a node parent node. Each node can be traversed down to its children or vice versa. Below we briefly describe some of these nodes using the TextFigure type in figure 2.5. Lets start with the declaration of the class `TextFigure`.

Type Declaration After parsing the `ICompilationUnit`, which is a representation of a Java file, the resulting node is of type `CompilationUnit` which a representation of Java source code. `CompilationUnit` is composed of a number of `AbstractTypeDeclaration` nodes. The node `TypeDeclaration` is a child of `AbstractTypeDeclaration`. The `TypeDeclaration` of `TextFigure` class is composed of a list of `BodyDeclaration` nodes. `BodyDeclaration` nodes are of three kinds:

- `FieldDeclaration` e.g. `(private int fOriginX;)`
- `MethodDeclaration` which represents a method. Method declaration is composed of the method header (access modifier, return type, name, parameters list and throw decla-

rations) and a body. `TextFigure` has two method declarations `removeBy(int, int)` and `changed()`.

- `TypeDeclaration` such as internally declared types: classes and interfaces. This kind of types are also known as member types. For example, the class `EventDispatcher`.

Method Declaration Each one of the method parameters is of type `SingleVariableDeclaration`. The body of a method declaration consists of a number of `ExpressionStatement` nodes and `VariableDeclarationStatement` nodes. Declared local variables in method body are of type `VariableDeclarationStatement`. Every other statement is of type `ExpressionStatement`.

`ExpressionStatement` is of many types. For example,

- if statement,
- assignment statement,
- method invocation, and
- postfix expressions.

Below we give some examples using code from figure 2.5

- The call to `willChange()` and `changed()` from `MoveBy(int,int)` is of type `MethodInvocation`. The call `super.changed()` from `changed()` method is of type `SuperMethodInvocation`.
- The if within the body of `moveBy()` is of type `IfStatement`. The conditional expression is of type `Expression`. Expression can be either an `InfixExpression`, `PostfixExpression`, `PrefixExpression` or `MethodInvocation`. The expression `(fLocator != null)` is of type `InfixExpression`. `InfixExpression` is an expression that is made up of two operands: left and right and an operator. The right operand of this expression (null) is of type `NullLiteral`.
- `SimpleName` node is a name of a field, method name, method parameter or local variable declared in a method. For example, in the method call `fLocator.moveBy(x, y)`, fLocator is a simple name of a field, x and y are simple names of method parameters and moveBy is a simple name of a method name.
- The expression statement (++y;) in method `moveBy(int, int)` is of type `PrefixExpression`. `PrefixExpression` is made up of one operand and an operator. Prefix operators are

Table 2.3: Node Type of the Expression of Method Invocation

Expression Node Type	Example
void	as in the call `updateLocation()` called from the body of the method `changed()`.
`ThisExpression`	as in `this.changed()` called from the body of the method `moveBy(int, int)`.
`SimpleName`	as in `fLocator.moveBy(x, y);` called from the body of the method `moveBy(int, int)`.
`MethodInvocation`	as in `getEventDispatcher().fireCommandExecutableEvent()`. The expression of the method call `fireCommandExecutableEvent()` is the method call `getEventDispatcher()`. The expression of `getEventDispatcher()` is null. This process is iterative until the expression is null, simple name or this expression.

- ++ Increment
- −− Decrement
- + Plus
- − Minus
- ~ Complement
- ! Not

- The expression statement (`x--;`) in method `moveBy(int, int)` is of type `PostfixExpression`. There are two postfix operators in Java. The increment (++) and the decrement (−−).

Method Invocation A method call is made up of an expression and a call. The expression is the receiver. This receiver can have one of the following cases shown in table 2.3.

This list of AST nodes is not comprehensive as we only explained the ones we need when explaining the following chapters.

2.4 Related Work

The research on aspect refactoring tools is divided into two phases: aspect mining and aspect refactoring. Aspect mining techniques aim at finding the candidates for refactoring step, which

```
1  public  class TextFigure {
3      private int   fOriginX;
       private int   fOriginY;
5      private AbstractCommand.EventDispatcher myEventDispatcher;
7      public void moveBy(int x, int y) {
       willChange();
9      ++y;
       x--;
11     if (fLocator != null) {
           fLocator.moveBy(x, y);
13     }
       this.changed();
15   }
17     public void changed() {
       super.changed();
19     updateLocation();
     }
21
       public void removeCommandListener(CommandListener oldCommandListener) {
23         getEventDispatcher().removeCommandListener(oldCommandListener);
     }
25
       protected AbstractCommand.EventDispatcher getEventDispatcher() {
27         return myEventDispatcher;
     }
29
       public static class EventDispatcher {
31     ...
       public void fireCommandExecutableEvent(){ ... }
33   }
35 }
```

Listing 2.5: TextFigure

performs the actual transformation of the candidate into aspect. Aspect miners are of two types: query-based or generative-based. Query-based aspect-mining approaches depend on some textual input from the user while generative-based ones are usually dynamic and generate the concern candidate based on the structural information taken from the source code. Because of the importance and the necessity of the mining stage, we give an intensive overview of aspect-oriented mining tools, although our work focuses mainly on automating the process of aspect-orinted refactoring and providing tool support for concern separation.

2.4.1 Aspect Mining Tools

Developers face difficulties understanding code pieces that interact to implement a concern and how different concerns interact with each other. To support developers in finding these pieces of code and identifying crosscutting concerns (CC), much research has tackled this problem and supported developers with a number of mining tools with the objective of finding those different pieces of the puzzle to start restructuring the code.

Query Based Approaches

AspectBrowser

AspectBrowser [19, 33] was initially implemented as a standalone tool until the discovery of the useful Eclipse plug-in framework; it has been retargeted to work as an Eclipse plug-in. AspectBrowser is a set of tools that help the developer focus on software modification tasks including Aspect Emacs and Nebulous. Aspect Emacs, a Lisp extension to GNU Emacs, represents each aspect as a pair of regular expression and a colour with the help of two external tools: redundancyfinder and aspectfinder, which both result in a set of candidates and their count of occurrences in the source code. Aspect Emacs allows for adding, deleting and annotating each candidate aspect. The Nebulous tool provides a global view that shows how the different elements that are part of a CC crosscuts the source code. Nebulous view shows files that are part of the aspect as a strip. The first row of the strip represents the first line of code in the file. If the same line has more than one aspect, Nebulous shows the row in red. When clicking any of the lines, the corresponding code get opened and highlighted in the Aspect Emacs view.

ConcernMapper ConcernMapper [31] maps program elements (e.g. field, method, statement) into a concern construct. A concern construct is a representation of a functionality that involves several programs entities including fields, methods, and statements. ConcernMapper is a plug-in implemented as a view where the user can create a concern and then use Java development tools to query for the concern's elements. Once an element is found, it then can be added by dragging and dropping it into the desired concern. In the case where searching results involves an element that

was already mapped, the name of the concern where it was mapped is displayed beside it. Other features of this plug-in include editing, deleting and modifying concerns; renaming a concern and moving elements between concerns. This plug-in was built with two main objectives: to support a developer with a tool to simplify software modification task and to provide a framework that allows plug-in to extend ConcernMapper through its Application Programming Interface (API) .

Feature exploration and analysis tool (FEAT) [30] is based on concern graphs, which are programming language, independent, mathematical models based on relational algebra. FEAT is composed of two views: concern graphs and participants views. Concern graph views show concerns and their sub-concerns and, when clicking any concerns, displays the concern's participants in the participants view. A participant is organized as classes and their members that are part of the concerns. Clicking on any participant displays its source code and selecting it shows how it relates (e.g. one method calls another) to other participants in the active concern in the relations view. The user starts the process of querying the program for concerns by creating an empty concern graph. FEAT internally builds a database for the whole selected project and creates a program model which is a set of relation labels (e.g. declares, calls, calledBy) over all the program elements (classes, interfaces and methods) which produces a set of relations.

For example, a relation on a class `A` with two methods `b` and `c` is specified as ({`A`,`b`}, {`A`,`c`}). FEAT uses 23 predefined relation labels. Selecting any element from any of Java views or even FEAT views, the user can query the program model by choosing from the tool-provided set of queries. The result of the queries is displayed in projection view. The query is internally represented as a projection over any fragment. To illustrate, a fragment is a domain, a label, a range and a projection i.e., (Domain, label, Range, projection). Using class `A` as an example, a fragment can be specified as ({`A`}, declares,{}, {`A`,`b`}, {`A`,`c`}). Clicking an element in the projection view displays its declaration in the editor, while selecting it shows its relation with the other elements (e.g. only the actual call to a method in the call's relation is highlighted) in the relation view.

As described, the FEAT plug-in was implemented as a complete perspective with five main views, which mainly assists the developer to identify the participating elements in implementing a particular concern. Since FEAT uses a stored database of the program, it has implemented a way to keep the stored model consistent with different versions of the program after modification.

Five case studies were performed to validate three different features of the concern graph: functionality, robustness and cost-effectiveness. Concern graph is functional if it eases the modification of a code that implements crosscutting concern. This is because FEAT allows the programmer to document the elements (e.g. methods) that are part of a concern. This eases the modification task. Also, FEAT is robust if the concern graph can identify concerns over multiple versions of a system. Applying FEAT on five Java projects showed that only one of them was robust and around half of

them were functional and cost-effective.

Generative Based Approaches

The Prism Aspect Miner The Prism Aspect Miner [36] uses a probabilistic approach that extends the page-ranking algorithm [24] to generate ranks for the popularity (frequency in which an element is visited by other elements) and significance (if an element references a large number of elements) of the program's elements based on the coupling graph of the underlying program. Finding the popularity and the significance is done randomly when a random element is chosen to start the algorithm. Running the algorithm creates the coupling graph with vertices referring to program's elements (e.g. method, package, class) and edges referring to the relations (calls, references, extends superclass, implements an interface) between various program elements. There is a high probability for a vertex to represent a crosscutting concern, if it is referenced by a large number of other elements or by elements that are more likely to be considered CC ones. Alternatively, there is a higher probability for an element to represent a core concern if it references a large number of other elements, or if it references elements that are more likely to be considered as core elements. In short, the algorithm selects the CC based on the natural order of their popularity and significance value.

Fan-in Analysis Tool Fan-in Analysis Tool [25], which is based on method fan-in, is composed of three steps to seek for CC "seeds". It first finds the number of callees (fan-in metric) for each method; second, it filters these results so only those that are most likely related to the concern remain; third, further analysis is made to the remaining methods to make sure only those parts of the concern implementation are reported in the result set. To calculate the fan-in metric the abstract syntax tree is built for the selected source and then the call graph is also generated. After calculating the fan-in metric value for every method, methods are ordered according to their metric value, and then a number of filters are applied; methods with metric value less than a user defined threshold, getters and setters and utility methods are eliminated. The analysis step follows several rules (e.g. calls always occur at the beginning or end of the method) to generate the final set of methods that are most likely to implement a concern. The Fan-in analysis plug-in uses the fan-in view to display the fan-in metric in which methods can either be ordered alphabetically or according to the fan-in value. The same view is used for the filtering step where the user has to supply the threshold value or some naming conventions to exclude callers or setters and getters methods from the result set. The user can inspect these results and manually add those that he thinks are part of a concern to the seeds view with the ability to annotate each seed. Applying this tool to a number of Java projects such as JHotDraw, PetStore and Tomcat server indicates that the tool can positively identify seeds with probability of 50% to 75%.

Dynamo Dynamo [34] uses formal concept analysis, a branch of lattice theory that can be used to provide meaningful groupings for a group of objects with shared attributes. Formal concept analysis takes a context as input, which is a boolean relationship over a large but finite set of objects O (e.g. Java, C++, Smalltalk, scheme) and their attributes A (e.g. functional, object-oriented, static typing). Given such context, formal concept analysis finds the maximal groups of objects and their associated attributes, known as concepts, such that all objects share the same attributes and those attributes should hold for every object in the concept, no other object outside the group has the same attributes, and no other attribute outside the group holds for any object inside the group. Each group or context has an intent and extent, which represent the boolean relation. Concept intent is the objective of the concept while the extent is the concrete representation of that intent (objective). For example, a language is "object-oriented" as intent will result in the set {Java, C++, Smalltalk} as the extent. Partial order set over all the concepts intent or extent produces the lattice. In the case of aspect mining, the lattice results from the execution traces for a number of scenarios (use cases) of a program where methods that share common attributes are grouped as concepts and then partially ordered to determine if a possible candidate implementing a concern calls for code restructuring. This can be derived from two situations: a concept is labelled with methods that belong to different classes or when a concept is labelled with many methods from the same class. Although the entire process of creating the lattice from the execution traces is semi-automatic, the inspection of the graph to confirm if the candidates calls for code restructuring is manual.

Given the background on aspect mining, we review current efforts in aspect refactoring of those entities found by mining.

2.4.2 Aspect Refactoring Tools

Refactoring tools can be divided into automatic and semiautomatic tools. In Eclipse extension platform, many aspect refactoring tools have been prototyped as plug-ins that manipulate the program's AST.

AJaTS

The AJaTs template based language [8] is an AspectJ transformation and code generation tool that provide refactoring templates to refactor general concerns such as persistence in Java programs, and AspectJ to AspectJ transformation to better structure the AspectJ code. AspectJ code generation and Java code transformation both require a template source, AspectJ/Java (AJ/J) source and a destination file where the template is matched against the AJ/J source and the result of the transformation is another program stored in the destination file. To determine if the template matches the AJ/J source, the AST for both files is traversed for type compatibility where each node

in the AJ/J source is checked against a AJaTS variables in the template source. AJaTS Refactoring is semi-automatic; the user interacts with a wizard where he can choose from those templates originally provided by the plug-in and also can add his own ones. Unlike AJaTS, AJRefactor does not use templates but refactors the program by analyzing its abstract syntax tree. Also, AJRefactor only refactors Java to AspectJ.

AspectRefactor

For this tool there was no supporting documentation other than a text file. This file basically states that AspectRefactor [29] performs special purpose refactorings with several different automated actions, the main ones being refactoring Java Authentication and Authorization Security part of the code. Unlike AspectRefactor, AJRefactor is a general purpose refactoring tool.

CRAFT

CRAFT [18] is built as an open infrastructure that can automatically refactor Java into Aspects. After a manual experience of refactoring two large Java systems (Prevayler and HSQL), CRAFT has identified 35 aspect-oriented patterns or code smells [16] that can be recognized in any system. Hence, they can be refactored using CRAFT. Each pattern was given a name, typical situation in which this refactoring is needed and the recommended action to be taken with a motivation example. CRAFT uses a trigger to specify these patterns, which are used to search for possible matches (footprint) in the source code. Triggers are of three levels: statement level (e.g. declaration, if statement), method level (e.g. method signature and return statement) and class level (e.g. a class implementing an interface that is part of crosscutting concern). Running a trigger against the code results in many footprints that activate a number of Craftlets which are a sequence of refactoring steps. A single footprint might activate more than a Craftlet, in which case the CRAFT runtime asks for a user interference either to select the appropriate Craftlet or to preview the code before the actual refactoring can take place. This process is iterative until no footprints activate any Craftlets. During this sequence and after a number of Craftlets have been executed, object-oriented refactorings might take place. This is because refactoring might open up an opportunity to recognise additional footprints. CRAFT supports developers with the ability to write their own triggers and Craftlets with the help of the CRAFT API to create the required sequence of refactorings. CRAFT uses a query based mining technique to identify refactoring candidates based on the CRAFT trigger language, which consists of a set of trigger constructs. A trigger is simply a text file written using trigger constructs which is then parsed and matched the source code. Trigger constructs follow some rules to accurately find matching footprints. CRAFT uses triggers as input to match the program code against but in AJRefactor the program is not matched against any input but its AST is analyzed instead.

Any refactoring operation requires an intensive analysis where the code to be refactored undergoes a number of precondition check to determine if the refactoring should take place or should stop and report an error message explaining the reason of failure.

JastAddJRefactoring

JastAddJRefactoring [15, 5] is one of the tools that take advantage of the analysis features of the compiler (JastAddJ). It has been implemented as a framework for extensible refactoring and was built by extending the JastAddJ compiler. Building it this way permits using compiler features, such as renaming analysis and data and control flow analysis, as building blocks providing a developer with reusable components that would ease implementing refactoring instead of implementing them from scratch. These primitive refactorings include renaming for all declarations (e.g. field, method, type etc.), extract method, and encapsulate field. Unlike AJRefactor, JastAddJRefactoring is not built as Eclipse plug-in but can be run under Eclipse.

The aspect-oriented migrator tool

The aspect-oriented migrator tool [9, 10, 12] migrates code into aspects through two steps: first, discovery and transformation; second, selection and refactoring. In the discovery and transformation step the TXL language was used to perform source-to-source transformation using grammar based rules. A rule is composed of a replace part, matching pattern; and a "by" part, the replacement. A pattern is composed of pattern variables, followed by their type (from the Java Grammar), and terminals. The matching is done at the AST level. The output of applying these is all the code areas matching the six different grammar rules as this tool performs six refactorings. In the selection and refactoring step, the algorithm selects which refactoring to apply in case the same code is marked with different refactorings. The selection is based on the efficiency of performing the refactoring on the base code and on the quality of the resulting aspect code. After deciding which refactoring to apply, the refactoring starts by creating advice and pointcuts for all refactorings identified in the selection. All pointcuts and advice are then merged into one advice if possible, using the pointcut abstraction refactoring. Aspect Migrator implements six refactorings:

- extract beginning/end of method/handler;
- extract before/after call, refactors code that is either before or after a method call;
- extract conditional, a conditional statement which controls the execution of the code to be refactored;
- pre return, code to be refactored is just before the return statement;
- extract wrapper, code to be refactored is part of a wrapper pattern; and

- extract exception handling.

These refactorings make an intensive use of statement reordering and extract method object-oriented refactorings. The total percentages of refactored code when Aspect Migrator was applied to four Java projects, i.e. JHotDraw, PetStore, JSpider and Jaccounting was 20% for begin/end refactoring, 16% for before/after call refactoring, 3% for conditional refactoring, 1% for pre return refactoring, 4% for wrapper refactoring and 14% for exception refactoring. The reduction of the code size was as follows: JHotDraw 5%, PetStore 4%, JSpider 3.7% and Jaccounting 16%. Unlike this tool, AJRefactor refactors programs in one step and does not include any source-to-source transformation.

These are the set of tools we have found since the beginning of this research in 2009 and there were newer tools prior to finishing of this thesis.

Other Related Work

More research interest was put into building refactoring tools that are aware of the existence of aspect code [11, 32, 35] such that whenever the Java code changes, the refactoring tool should account for the possibility that changing the code might change a point cut. Specifically, it might exclude or include some join points that were/were not previously part of the existing pointcuts. In [21] a method to rejuvenate pointcut expressions was implemented.

Several efforts have identified different patterns that frequently occur in object-oriented and aspect-oriented programs. For example, CRAFT [18] has identified 35 different patterns which they implement to refactor any code that calls for restructuring. In terms of design patterns, they can be refactored using these small refactorings [26] but this means for a pattern such as the observer the user has to analyze the code to find pattern participants, identify the steps and the corresponding code changes that would transform the pattern into aspect. AJRefactor tackles the issue of identifying and analyzing pattern elements and if the refactoring can take place.

2.5 Summary

The implementation of design patterns in Java presents some issues; modularity being one of them. Some patters implementation tend to spread over multiple classes resulting in a code that is difficult to understand, read, and maintain. Carrying part of the pattern implementation hinders the cohesion of the classes. The solution we propose is to separate the primary functionality of the classes from the crosscutting behaviour using aspects.

We contribute an Eclipse refactoring plug-in (AJRefactor) to refactor the observer pattern implemented in Java to AspectJ. Since many people have explored the area of aspect refactoring

tools, we review related work along with a brief description of how our work differs from each one of them.

Overall, our goal is to show the power of aspects and aspect constructs that AspectJ offers to enhance the modularity of Java programs. Following chapters dive into the details of AJRefactor implementation and design. Furthermore, we discuss the results of the case studies of applying AJRefactor on two Java projects.

Chapter 3

AJRefactor

3.1 AJRefactor Functionality

Before diving into design decisions and implementation details of AJRefactor tool, it is valuable to understand its use. First, we look at the pattern implementation in Java. Second, we explain the transformation steps to better modularize this code using aspects. There are mainly two different observer implementations we refactor with AJRefactor. One where the pattern is implemented following the pushing technique and one where observers observes a change and then performs an update by calling their update methods. Below we explain the refactoring details of each one of these cases.

3.1.1 Observer Pattern Following Pushing Technique

The original program in figure 3.1 shows a sample implementation of the observer pattern. It has four classes:

- the `Point` class represents the subject. `Point` class declares three fields: x, y co-ordinates and a list to store the point observers. It also declares a number of setter and getter methods.
- the `Screen` class represents the observer. The `Screen` displays these points.
- the `Subject` interface. This interface allows the `Point` to add and remove and notify the observers through its three methods: `addObserver`, `removeObserver`, and `notifyObservers`.
- the `Observer` interface. This interface allows the `Screen` to update itself, when notified of a change in a point state, by implementing the `refresh` method.

The `Screen` needs a mechanism to update its display and relocate those points once they move. For that, the screen needs first to register itself as one of the point observers using the point's `addObserver` method. Second, it has to implement the observer interface update method that gets called by the subject to notify its observers of a change. Now, every time the point sets its co-ordinates it calls its `notifyObservers` which loop s over the list of observers and call their

refresh method. The refresh method executes the action related to each observer which is in this sample program the redisplay action of the screen.

```
public class Point implements ChangeSubject{
     private HashSet<ChangeObserver> observers;
     private int x;
     private int y;

     public int getX() {
          return x;
     }

     public int getY() {
          return y;
     }

     public void setX(int x) {
          this.x = x;
          notifyObservers();
     }

     public void setY(int y) {
          this.y = y;
          notifyObservers();
     }

     public void addObserver(ChangeObserver o) {
          this.observers.add(o);
     }

     public void removeObserver(ChangeObserver o) {
          this.observers.remove(o);
     }

     public void notifyObservers() {
          for (ChangeObserver e1: observers)
               e1.refresh(this);
     }
}
```

```
public class Screen implements Observer {

     private String name;

     public void display(String s) {
          System.out.println(name + ": " + s);
     }

     public void refresh(Subject s) {
          String subjectName = s.getClass().getName();
          display("update received from" + subjectName);
     }

}
```

```
public interface Subject{

    public void addObserver(ChangeObserver o);

    public void removeObserver(ChangeObserver o);

    public void notifyObservers();
}
```

```
public interface Observer {

    public void refresh(Subject s);
}
```

Figure 3.1: Java Observer Program Before Refactoring

Although this is one of the cleanest ways to implement this pattern in Java, it still can benefit from more modularization such that the observers can still be updated without the point being responsible for the notification. We explain how we achieve clearer code with all the notification and updating oblivious to the `Point` and `Screen`. We explain refactoring details shown in figure 3.2.

The yellow part of the figure has two aspects: `CoordinateObserver` and `ProtocolAspect`. We refer to the former as the instance aspect and the latter as the protocol aspect.

Instance Aspect

This aspect captures each observer relationship. For example, the screen redisplay itself when the point sets its x. We have two actions: the point moves and the screen redisplays itself. This is implemented in the `Coordinate` aspect as follow:

- We declare `Screen` to implement `Observer` and `Point` to implement `Subject`
- We declare the `subjectChanged` pointcut to capture the call to `setX` join point.
- We add the `refresh` method declaration to this aspect as inter-type declaration of `Screen`.

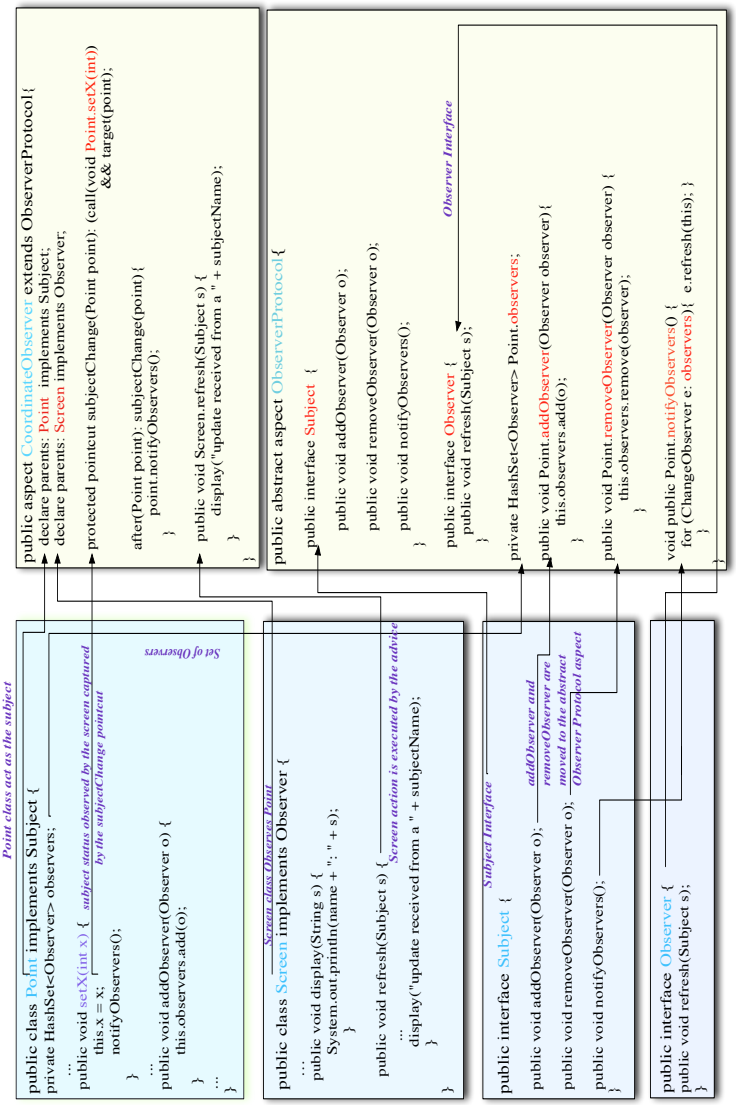

Figure 3.2: Aspect Refactoring for the observer pattern

- We delete the call to notify from the `setX` method and move it into the aspect so that it get executed as the advice.

This would be enough to capture this specific instance of the pattern but we want the pattern to disappear completely from the point and screen classes. Therefore we declare the protocol aspect.

Protocol Aspect

This aspect declares the parts that are common to every observer and every subject. It declares two interfaces `Subject` and `Observer`. It declares the methods that manipulates the list of observers: add, remove and notify. Finally, it declares a field to store the list of observers. This aspect is abstract. Each instance aspect of the pattern should extend this aspect. In our example this aspect

is created gradually during the refactoring.

- The point implements the `Subject` interface. We declare this interface internally in the protocol aspect.
- We move `add`, `remove` and `notify` observers methods from the point to the protocol aspect.
- We remove the `Subject` interface implementation from the point.
- We move the list of observers field declaration into the aspect as inter-type declarations on `Point`
- The screen implements the `Observer` interface. We move this interface and its operation into the protocol aspect; we declare this interface in the protocol.
- We remove the observer interface implementation from the `Screen`.

The last remaining step is to go back to the instance aspect and make the instance aspect a sub-aspect of the protocol. The purpose of having the declaration of the subject and observer interfaces moved into the protocol aspect is to completely isolate the observer pattern implementation into aspects.

At this point the transformation is complete. If we want to capture the observer relationship between the point and the screen when point sets its y co-ordinate, we only need to create a new instance aspect similar to the `CoordinateAspect` and make the pointcut captures the call to `setY` as in figure 3.3. The protocol aspect needs to be created once and then reused for each instance aspect. The same figure shows that the only coupling remaining is within the advice as the method `notifyObserver()` get called only within the after advice. This method get called after any of the two methods `setX(int)` and `setY(int)` get executed.

This aspect could be broken into two different aspects where the first has the `Observer` and `Subject` interfaces and the second declares the different methods of the `Point` class as inter-type declarations but we have created it this way for simplicity.

Declaring `AspectOfSetY` as privileged is not needed at this aspect but this is a design decision. Instead of increasing the overhead by examining each method declared as inter-type declaration to see if it references any methods or fields that require privileged access we decide to declare every instance aspect as privileged.

The parent declaration of the `Screen` class is not needed in the aspect `AspectOfSetY` as we have already declared it in the coordinate aspect. This is a design decision because it is possible for one subject to have multiple observers that might watch for different changes in the subject state. For each instance aspect we would declare the parent of the observer and whether it is a direct implementor of the observer interface or it implements an interface that extends the observer

interface. The same apply for the inter-type declaration of the method that the observer implements. These are some of the limitations of our tool. We explain these in details in the design chapter 4.

```
public privileged aspect AspectOfSetY extends ObserverProtocol{
    declare parents: Screen implements Observer;

    public void Screen.refresh(Subject s) {
     display("update received from the subject");
    }

    public pointcut subjectChange(Point point):
     call(void Point.setY(int)) &&
     target(point);

    after(Point point): subjectChange(point){
     point.notifyObservers();
    }
}
```

Figure 3.3: Coordinate Aspect of setY

3.2 AJRefactor GUI

Next, we turn our attention to the user-level view of our tool. Figure 3.4 shows the Graphical User Interface (GUI) of AJRefactor. Our tool adds a new menu item called AJRefactor that has one sub-item named Refactor Observer. AJRefactor is a separate menu because the existing refactor menu is built into the base Eclipse system using actions and we did not want to change that core. Our menu instead in incorporated using the command extension point facility. When a developer selects our new menu item, our refactoring interface is displayed as in figure 3.4.

- The process starts when the programmer selects the statement that changes subject state and presses the refactor observer menu. If the selection changes a field in the selected class, AJRefactor considers the class where the selection occurs the subject.

- Calculating the types of the subject class fields, types of the parameters of the subject constructor and types of the parameters of the method that changes the state of the subject, AJRefactor populates the list of potential observers and presents them in the first page of the wizard. The wizard pops up to the user only when the initial conditions checking phase

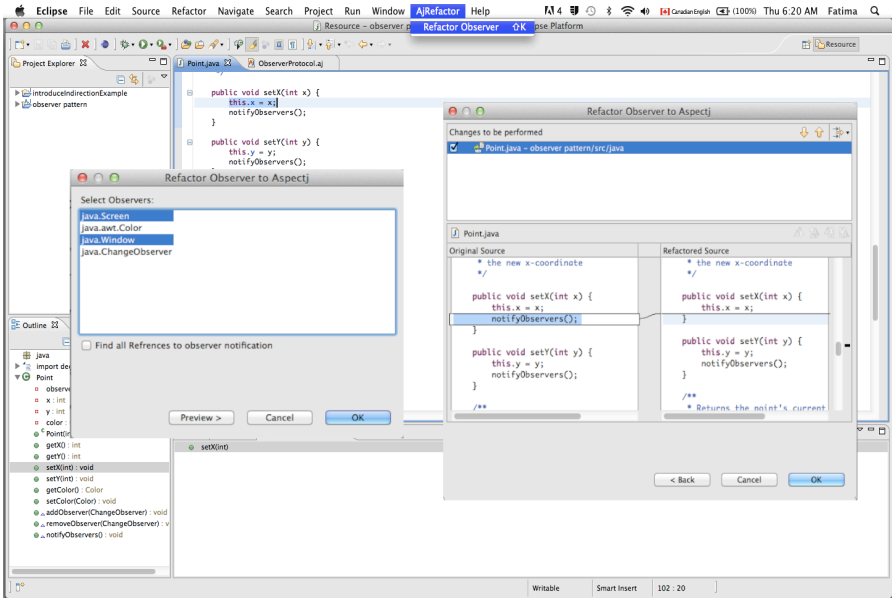

Figure 3.4: AJRefactor GUI

passes successfully. We explain the details of the initial conditions checking phase on the AJRefactor implementation section 3.3.

- The user then can select the actual observers. After calculating the observers, AJRefactor counts the number of statements that follow the selected expression. Here we distinguish four different cases.
  - If it is one and of type a method call, it retrieves the method binding to find the type of the class defining that method and whether it is a super class, super interface or the subject class itself. This is used in identifying the other components of the pattern.
  - If it is more than one this would turn into the second type of the observer pattern that we handle in our tool. We call this observer update case. We explain this in the next section 3.2.1.
  - If there is only one statement whose type is not method invocation. For example, an assignment statement. The refactoring would terminate indicating that there is no observer instance. This is also the case when there is no statements after the selected expression.
  - There might be some cases where the programmer has to do some changes whether manual or automatic for the refactoring to take one of the shapes that could be refactored with AJRefactor. For example, when the method is not well modularized as it changes the subject state and then loops over the list of observers and notify them of the change.

To fix this, the programmer has to perform an extract method refactoring before applying AJRefactor. Now, AJRefactor deals with the first case.

- In all cases, AJRefactor tries to find the method declaration of the method that notifies the observers and examine its body closely. This method calls the observer update method where the observer updates itself in response to the subject change.

- The wizard provides the user with an option to find all the pattern instances in the subject.

- Once the tool collects all the pattern parts and the final condition checking passes successfully, the user can preview code changes or just press the finish button.

- Once the user confirms the changes to the participating classes, AJRefactor starts rewriting the whole pattern into aspects. Specifically,

  - It moves the operations that manipulate the list of observers (i.e. registering and unregistering them from observing the subject and notifying them of any change to the subject state) if they exist into the abstract aspect. If these operations do not exist, the refactoring still continues as intended as these operations are not required for the refactoring to complete.

  - It also captures the call to the method that changes the subject state in the pointcut, and

  - Executes the call to notify in the advice.

- By creating these aspects the refactoring is considered complete and the change can effect the workspace. The user can now run tests to verify correct behaviour.

3.2.1 Observer Update

This type of observer pattern is very simple. Observers watches a change in the subject state that when happens observers performs an update in their state too. This is done by immediately invoking their update operation. Figure 3.6 shows an example with steps to refactor it using AJRefactor. After a number of checks and calculations AJRefactor determines that it is dealing with observer update pattern. AJRefactor displays these update method invocations to the programmer. She selects the ones involved in the update. The selected invocation are marked for deletion from the subject class. Then, AJRefactor creates an aspect where these invocations get executed in the advice. The pointcut is created based on the location of the update invocations in the method that changes the subject state. Details of the heuristics used are given in our design chapter 4.

3.3 AJRefactor Implementation

There are three required components to implement a refactoring:

1. a refactoring action,
2. a refactoring wizard,
3. and a refactoring class.

To describe AJRefactor implementation we first start by explaining the general functionality of each of three components. Then, we describe our implementation of the pattern which we divide into two based on the type where user selection occurs. That is either the subject or the observer class.

3.3.1 Refactoring Action

Refactoring actions are used to launch the tool after listening to user selections. It also enables the refactoring based on the changes to the selection provided by the workbench selection service. AJRefactor contributes a menu to the workbench using the command framework. The choice was to contribute a menu next to the workbench window menu such that it activates only when Java editor is active. In case many menus contributes to the workbench main menu, this would clutter the user interface but using perspectives the user can manage which menus can appear.

When the user selects some text in the Java editor and pushes the refactoring menu button, user selection is validated. Based on the selection, AJRefactor constructs the prober refactoring class. Once the refactoring object has initiated, AJRefactor would check for a number of a initial conditions. If these conditions pass successfully, the refactoring wizard pops up waiting for user input, otherwise the refactoring terminates and a message pops up indicating the reason of failure.

3.3.2 Refactoring Wizard

Our ObserverWizard refactoring wizard extends `org.eclipse.ltk.ui.refactoring.RefactoringWizard` and provides the facility to implement the error pages, input pages and preview pages that should be of type `RefactoringWizardPage`. It also provides the facility to add pages to the wizard based on the user input. The first page displayed for the user depends on the number and type of statements that come after the selection. This analysis is done in the initial conditions checking phase. This page can be either one that asks the user to select from the list of potential observers, one that asks the user to select from the list of potential subjects, or one that asks the user to select the method invocations that are part of observe update. Beside the list of potential observers, the user can choose to find all notifications to the observers. The preview

button is activated when the user selects at least one observer, one subject or one update operation. Clicking this button activates the final condition checking that finds all other pieces of the pattern and calculates the changes to be performed for the refactoring to take place.

3.3.3 Refactoring Class

My RefactorObserver refactoring class implements `org.eclipse.ltk.core.refactoring.Refactoring` and provides the core functionality for the refactoring through its three main methods:

- *checkInitialConditions:* checks the refactoring initial conditions and determines if the refactoring wizard can appear to the programmer in case the refactoring implements one.
- *checkFinalConditions:* this method is invoked after the initial conditions passes. It checks programmer inputs and does further calculations.
- *createChange:* this method is invoked after the final condition passes. It calculates and performs the required changes to the compilation units effected by the refactoring.

We describe each below.

Initial Conditions Checking

The method `checkInitialConditions` takes `IProgressMonitor` as a parameter and returns `RefactoringStatus` which indicates whether the conditions checking passed or not. `RefactoringStatus` is of five kinds:

- `RefactoringStatus.ERROR`,
- `RefactoringStatus.OK`,
- `RefactoringStatus.INFO`,
- `RefactoringStatus.WARNING`, and
- `RefactoringStatus.FATAL`.

Failing to satisfy the following conditions will result in a fatal refactoring status that immediately terminates the refactoring with a message indicating the reason of failure. Below are the checks we perform in all refactoring classes regardless of the starting type i.e. subject or observer type.

- The underlying `ICompilationUnit` is error-free as the refactoring framework cannot change a type that has compilation errors. Also, calculating the binding of fields, methods and types in that compilation unit may fail.

- The project can be converted into AspectJ project. The tool refactors the observer pattern to aspects which requires the project to have AspectJ nature enabled. The tool adds this capability to the project using Java API [14] which might fail for reasons such as the AspectJ plug-in is not installed.

Assuming the initial condition checking has passed, `checkInitialConditions` returns a `RefactoringStatus.OK` and the refactoring framework calls the `checkFinalConditions` method.

Final Conditions Checking

Similar to `checkInitialConditions`, `checkFinalConditions` takes `IProgressMonitor` as an argument and returns a `RefactoringStatus` indicating whether the condition checking has passed or failed. The Refactoring class calls this method after the initial conditions check passes and after the user provides the necessary data. It validates user inputs and collects some important data for the change generation step. This method can be called more than once according to the interaction with the user-interface, but it has to be called after the initial conditions checking and before the create change step. This method also calculates the change in the classes participating in the pattern. During change calculation if a `RefactoringStatus.FATAL` has returned, the programmer is presented with the unsatisfied-precondition checking-message; otherwise the method `createChange(IProgressMonitor)` is called.

Following are some of the conditions that AJRefactor perform and what might cause them to fail and terminate the refactoring.

- *Creating aspect files* Creating the aspect file is performed for all refactoring instances regardless of the starting type i.e. subject or observer. Creating these files is done by examining the access modifier of each one of the observer files. If they are publicly accessible, AJRefactor creates a new package for the newly created aspects including the protocol one. If any of the types is only accessible within its package, the aspect is created within its package. Now, if both types are in two different packages and both are accessible within their own package, AJRefactor terminates the refactoring with a message. For example, the message is *"one of the types participating in the refactoring instance is inaccessible within the aspect, please change the access modifier for the refactoring to proceed"*

- *Finding pattern participants* AJRefactor depends on Java search engine to find pattern components. This engine might through `JavaModelException`. We handle this exception by terminating the refactoring which a fatal error refactoring status. The refactoring status takes a string which the message we present to the programmer after termination.

- *Registering changes to compilation units of affected classes* Registering and rewriting changes to the classes affected by the refactoring might also fail. Failing at this stage also terminates

the refactoring and produces an error message. For example, *"There was an error while registering changes to the Foo class"*

Creating the Change

This method is called after the final conditions check passes successfully. It returns an object of type `org.eclipse.ltk.core.refactoring.Change` that is used later by the refactoring user-interface to generate the change preview. We use the same change object to rewrite the classes affected by the refactoring once the programmer confirms these changes. We accomplish this by applying the `ASTRewrite` on each one of these classes. The refactoring terminates after the change applies to the workspace. If the change object is not empty, we create the different aspects.

According to the current state of the AspectJ plug-in, it was not possible to create an aspect from scratch using AspectJ AST API. This is because these APIs are still under development. Therefore we copied and modified the code that AspectJ development tools use when users need to create a new aspect using the create new aspect wizard. This involves referencing some of the classes that are used internally and not part of the public API. This is one of the places where the plug-in needs to improve.

Similar to Java classes, aspects are created using the `org.eclipse.jdt.core.IType` class. We add pointcuts, advice and inter-type declarations using the different methods provided by the `IType` class. Pointcuts and advice are added with `createMethod` while inter-type declarations are added using `createField` method. One of the arguments that these methods take is a string representing the declaration of the element to be created.

Figure 3.5 summarizes the implementation details of the refactoring class methods: `checkInitialConditions`, `checkFinalConditions` and `createChange`.

We divide the implementation of AJRefactor into two: pattern refactoring starting with subject type and pattern refactoring starting with observer. We also refactor the pattern implemented with the common Java API (`java.util.Observable` and `java.util.Observer`) whether the starting type is the subject or the observer. Below we describe the implementation of each one of these refactorings through the three components that we have described earlier. These components are refactoring action, refactoring wizard and refactoring class.

3.4 Refactoring Observer Pattern Starting with Subject

We explain the implementation through the three components of an eclipse refactoring.

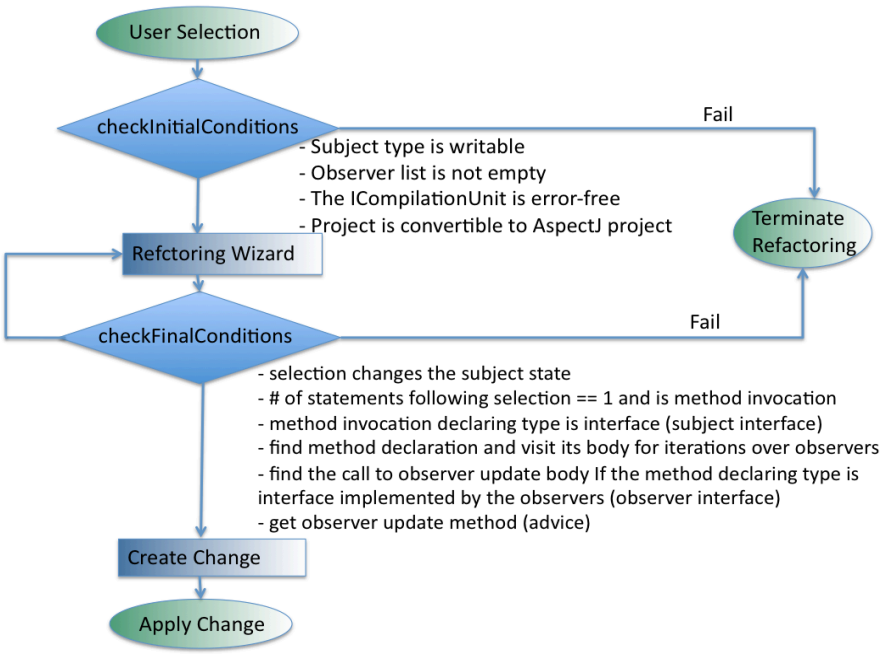

Figure 3.5: Summary of the Refactoring Class Methods Implementation

3.4.1 Refactoring Action

When the user selects some text in the Java editor and pushes the refactoring menu button, user selection is validated. AJRefactor constructs the refactoring instance based on the selected node type. If the selection is one of the following cases, AJRefactor considers the type where the selection occurs is the subject and the refactoring process continues searching for observer and the other components of the pattern.

(1) The node is a method invocation whose declaring type is the same type where the selection occurs or any of its super classes or any of its member classes.

(2) The node is a method invocation where the expression is a simple name of a field, `this` expression, or null.

(3) The node is a statement of type super-method invocation. For example, `super.foo()`.

(4) The node is an assignment statement where the left side is a field.

(5) If the operand of a postfix expression is a simple name of a field.

(6) The node is an if-expression and the expression

- is a method invocation as in 1,2 or 3.
- is an operand of an infix, postfix or prefix expression whose node type is a `SimpleName` of a field, this expression or a method invocation as in 1,2 or 3.

If the selection was evaluated to be one of the aforementioned cases, AJRefactor would check if the observer is implemented using Java API. To check for this, AJRefactor checks if the subject extends the `java.util.Observable`. If it does not, AJRefactor continues as we describe below.

Once the refactoring object has initiated, AJRefactor would check for a number of a preconditions. If these conditions passes successfully, the refactoring wizard pops up waiting for user input, otherwise the refactoring terminates and a message pops up indicating why the checking fails. By this we have the first component of the pattern that is the subject class. From this, the refactoring can analyze the statements after the selection. Based on the number and type of theses statements AJRefactor either presents the user with list of potential observers or with list of observer update calls.

3.4.2 Refactoring Wizard

- If the subject change is followed by a single method call whose explicit or implicit receiver is `this`, the wizard presents the programmer with a list of potential observers.
- If the subject change is followed by a one or more method calls whose receiver is not `this` whether explicit or implicit, AJRefactor presents the user with list of observer update calls.

3.4.3 Refactoring Class

Initial Conditions Checking

Aside from ensuring the project has AspectJ capability and the subject type is compiler-error free, AJRefactor checks for the following conditions. Failing to satisfy any of them results in a fatal refactoring status that immediately terminates the refactoring with a message indicating the reason of failure.

- Subject type is writable. This is essential because the refactoring cannot modify source code of a file that is read-only or binary. A read-only type is a type whose underlying source code is not modifiable. This is the case for `.class` files or files that are part of a zip or a jar. In this case the refactoring needs to delete or add nodes to the abstract syntax tree of the subject when the refactoring take place. An example of a type modification would be deleting the update call from the method that changes the subject state.
- There is at least one observer in the list of potential observers. Potential observers list is calculated from three different sources:
  1. subject class fields,
  2. parameters of user selected method, and

3. parameters of the subject constructor.

In all of these cases there might be some added complexity when fields and parameters type is parameterized. In this case we check if the parameter type is an interface type (e.g. field of type `HashMap<T>`). If it is we find interface implementers and their subclasses. The process is iterative if any of the implementors is an interface. The implementors and their subtypes get added to the list of potential observers. This supports situations where the subject has multiple observers that are stored in some sort of data structure (e.g. HashSet, Set, Array) whose parameter type is an interface implemented by observers.

- The type of statement that comes after the selection is a method invocation. If there is no statements after the selection or the there is one but it is not of type method invocation, AJRefactor terminates the refactoring and presents the user with a message stating that there is no observer pattern.

- If a single method invocation whose declaring type is the subject or any of its super classes, AJRefactor continues finding pattern participants. If a single method invocation where the receiver is not `this` or if there is more than one method invocation, AJRefactor considers this an observer update case where the user need to interfere to select the actual updates. Figure 3.6 shows an example of this type of observer pattern.

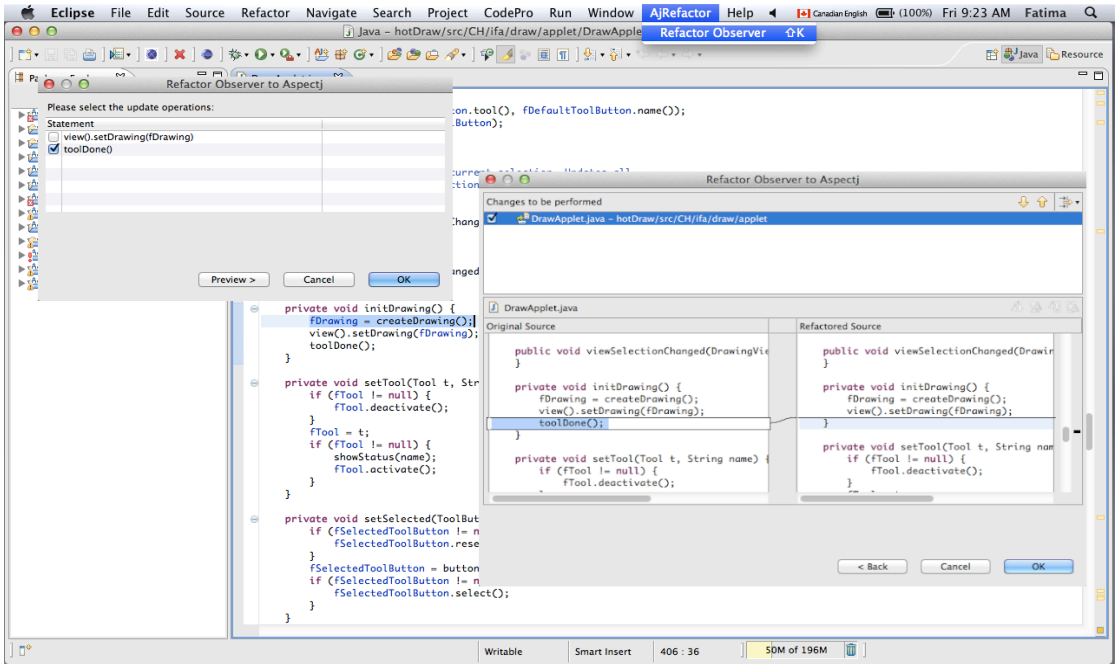

Figure 3.6: Aspect Refactoring for the observer pattern

Assuming the initial condition checking has passed, `checkInitialConditions` returns a `RefactoringStatus.OK` and the refactoring framework calls the `checkFinalConditions` method.

Final Conditions Checking

We refer to the example code in figure 3.2 to illustrate the following points as needed.

- *update pointcut* When the user selects the code that changes subject state, the method that encloses the selection is the one that is intercepted in the pointcut.

- *subject interface* The tool first checks the number of statements after the statement that changes the subject state; in this case, the `x` value. If it is one then it checks the type of that statement. If it is a method invocation then AJRefactor resolves the method binding and determines the declaring class. If the declaring class is the subject; in this case, the `Point` class, AJRefactor finds if it is an implementation of an interface type. This is because the interface type disappears during class loading. Therefore, we needed a way to find out if the method that notifies observers is an implementation of interface operation. Eclipse offers a general search engine that can be used to search the different projects defined in the workspace for java elements such as method references, field declarations and interface implementors. Using this engine we search for types that declares the update method; in this case the `notify` method, and then check if the subject type implements any of these resulting types. If so, we have located the second component of the observer pattern that is the *subject interface.* Algorithm 1 explains the process of finding the subject interface.

Algorithm 1 find the subject interface

```
if countOfStatementsAfterSubjectChange = 1 then
  if statementType = methodInvocation then
    if methodBinding = subjectType then
      results ←findTypesDeclaringMethod
      for all result in results do
        if isInterfaceType(result) and doesSubjectImplementsType(result) then
          return result
        end if
      end for
    end if
  end if
end if
```

- *Observer Interface* Retrieving the body of the `notify` method, we examine its code closely. The whole process of finding the components of the observer is done by traversing the AST and examining its nodes. Applying the visitor pattern, AJRefactor examines the type of each node. In most cases the body iterates over the list of observers i.e. the body can be one of following cases.

  – For loop. For example,

```
for(Iterator e = observers.iterator(); e.hasNext();){
            ((ChangeObserver)e.next()).refresh(this);
        }
```

Listing 3.1: Notify body - For loop

  – Enhanced for loop. For example,

```
for(ChangeObserver obs: observers){obs.refresh(this);}
```

Listing 3.2: Notify body - Enhanced For loop

  – While loop. For example,

```
Iterator<ChangeObserver> obs= observers.iterator();
        while (obs.hasNext()){obs.next().refresh(this);}
```

Listing 3.3: Notify body - While loop

  – Do-while loop. For example,

```
Iterator<ChangeObserver> obs= observers.iterator();
if(obs.next != null){
    do{ obs.next().refresh(this);} while(obs.hasNext())
}
```

Listing 3.4: Notify body - Do-While loop

We first check for generic types included in the iterator. If it is an interface then we examine if the actual observers selected by the user implements that interface. If no generics are used, then we check the body of the loop. In particular, we look for the receiver of the method invoked within the loop body. If the receiver type is an interface implemented by the actual observers, then we consider this type the *observer interface.*

Finding the observer and subject interfaces is not essential as they might not exist. It is possible for the subject to have only one observer and immediately call its update method. Another possibility is that the subject notify method is provided by one of the subject super classes or it is one of its own methods or it overrides this method. AJRefactor accounts for those cases as the two most important pieces are the method call that would be intercepted in the aspect pointcut and the update method of the observer that the advice executes.

- *Observer Update Method* This method is called within the body of `notify`. Using Eclipse search engine we search for classes that declare this method. The result is a list of methods. We examine each method for declaring type and if it is an actual observer, one of the types selected by the programmer, we store the method declaration. This declaration is used to create inter-type declaration when creating the aspect for each one of these observers.

- *Observers List Field* From the body of the notify we find the observers list depending on the type of the loop. We use the visitor pattern to visit the loop initializers and body to find the observer list. The process of finding the observers list depends on the type of the loop statement.

  - *Enhanced for loop* The enhanced for is composed of three parts. Listing 3.2 shows these components which we explain below.

    * a parameter which comes before the colon (e.g. `ChangeObserver obs`),
    * an expression which comes after the colon (e.g. `observers`),
    * and a body (e.g. `obs.refresh(this);`).

    We check the expression node type. If it is a simple name of a field, we retrieve its type binding and if it is one of the following cases we consider this simple name the observer list.

    * binding is the subject class.
    * binding is the subject interface.
    * binding is a member type of the subject.
    * binding is a parametrized type whose parameter type is the observer interface (if parameter type is provided).

  - *For loop* For statement is composed of four parts. The first three parts are separated by the semicolon. Listing 3.1 shows these components which we explain below.

    * The initializers (e.g. `Iterator e = observers.iterator()`). This initializer is of type `VariableDeclarationExpression`. It is possible to declare more than one variable at the same time. These declaration are of type `VariableDeclarationFragment`. For example, `int x=0, y=3;`. Fragment is made up of a

SimpleName and an initializer. The initializer node is of type Expression. Expression can be but not limited to one of the following cases.

· NullLiteral

· MethodInvocation

· ThisExpression

If the initializer node is of type method invocation whose expression is a simple name of a field, we retrieve its type binding. If the binding is one of these bindings we have mentioned in the enhanced for loop case, we consider this simple name the observers list.

* an expression (e.g. e.hasNext())
* an updaters. In the example there are no updaters but increasing (x++) or decreeing (x--) an int is an example of an updater.
* a body

– *While loop and Do-while loop* If any of these two statements were found, we check the node proceeding them looking for a node of type VariableDeclarationStatement which represents the declaration of an iterator over the list of observers. As VariableDeclarationExpression might declare many variables, the VariableDeclarationStatement might also declare more than one variable at a time. Each declaration is of type VariableDeclarationFragment. As we mentioned earlier in the for loop case, fragment is made up of a SimpleName and an initializer where the initializer node is of type Expression. Similar to the for loop, if the initializer node is of type method invocation whose expression is a simple name of a field, we retrieve its type binding. If the binding is one of these bindings we have mentioned in the enhanced for loop case, we consider this simple name the observers list.

- *Changing Patterns Participants* Eclipse provides an ASTRewrite class which records changes to the compilation units participating in the pattern implementation and then applies these changes back to the underlying files. The changes depends on the type of the observation relationship.

  – For observer instances implemented using the pushing technique, AJRefactor register the following to the ASTRewrite

    * removing the invocation to the notify from setX(int x),
    * moving the observers list and all methods that manipulates it into the protocol aspect,
    * moving the method that notifies observers to the protocol aspect,

    * moving observer update method into the aspect, and finally
    * replace the `implement` phrase for both the subject and observer interfaces with inter-type declarations.
  - For observer update case AJRefactor only removes the calls selected by the programmer from the method that changes subject state.

During change calculation, if a `RefactoringStatus.FATAL` has returned, the programmer is presented with the unsatisfied-precondition checking-message; otherwise the method `createChange(IProgressMonitor)` is called.

Creating the Change

This method is called after the final conditions check passes successfully. It returns an object of type `org.eclipse.ltk.core.refactoring.Change` that is used later by the refactoring user-interface to generate the change preview. We use the same change object to rewrite the classes affected by the refactoring once the programmer confirms these changes. After the programmer confirms these changes, AJRefactor creates the different aspects. If the observer pattern is implemented using the pushing technique, AJRefactor creates a protocol aspect to capture the list of observers and the methods that handles them including adding, removing and notifying the observer and an instance aspect for each observer. For the case where there a number of observer update invocations, AJRefactor would create one aspect with a pointcut and advice to execute these update calls.

3.5 Refactoring Java API Observer Pattern Starting with Subject

We consider this case a variation of the observer pattern starting with subject. Figure 3.7 shows an example.

3.5.1 Refactoring Action

If AJRefactor examines user selection and finds out it is one of the cases where the selected type is the subject type, AJRefactor performs one further check. AJRefactor checks if the pattern is implemented using common Java API. AJRefactor does this by checking if the subject extends the `java.util.Observable` type from the Java utility. If it does, AJRefactor constructs the refactoring instance that handles the refactoring of the pattern implemented using Java API starting with subject type.

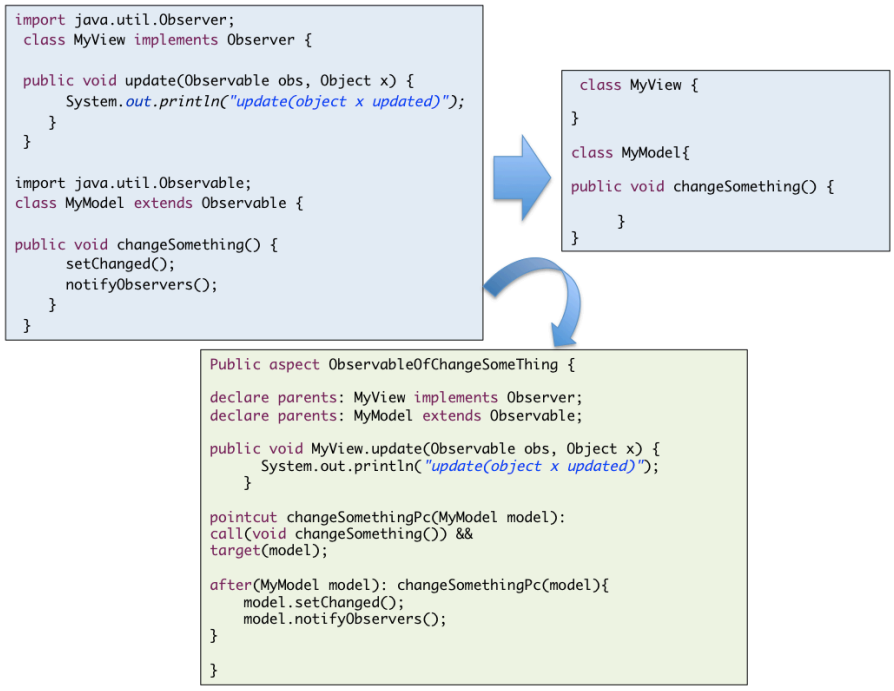

Figure 3.7: Observer Pattern Implemented with Java API

Once the refactoring object has initiated, AJRefactor checks for a number of a preconditions. If these conditions pass successfully, the refactoring wizard pops up waiting for user input, otherwise the refactoring terminates and a message pops up indicating the reason of failure. By this we have the first component of the pattern that is the subject class.

3.5.2 Refactoring Wizard

After calculating the types that implements `java.util.Observer`, AJRefactor presents them to the programmer to select the actual observers of the subject.

3.5.3 Refactoring Class

Initial Conditions Checking

AJRefactor performs the same checks it performs for the non-API observer pattern starting with subject.

Final Conditions Checking

In addition to the checks we perform when initiating the pattern, this method performs the following:

- *Method Invocations of Observable* AJRefactor finds the method invocations of `setChanged()`, `notifyObservers(Object arg)` or `notifyObservers()`. These invocations get executed by the advice in the aspect.

- *Rewriting Subject* When calculating the change in the subject, we check if there any references to `java.util.Observable` other than this in the method where user selection occurs. If no references are found, AJRefactor removes `Observable` class references from the subject. This includes

  - removing `Observable` from the import declarations,
  - removing `Observable` from the list of the classes the subject extends, and
  - removing invocations to `Observable` methods from the method that observers observe.
  - it is possible that `Observable` methods that observers use to register or unregister themselves from observing the subject are called any where in the program. These methods are Although registration of observer methods is not shown in figure 3.7, they can still be refactored. As is the case with the non API pattern. The methods deleteObservers() and `addObserver(Observer)` can be refactored as an observer update method case.

- *Rewriting Observers* for each observer, AJRefactor moves the implementation of `java.util.Observer` into aspect. This includes

  - removing the `Observer` from the list of interfaces the observer implements and replace it with an inter-type declaration in the created aspect.
  - removing the `Observer` from the list of import declarations.
  - moving the `update(Observable, Object)` declaration into aspect. AJRefactor creates inter-type declaration in the declared aspect.

During change calculation if a `RefactoringStatus.FATAL` has returned, the programmer is presented with the unsatisfied-precondition checking-message; otherwise the method `createChange(IProgressMonitor)` is called.

Creating the Change

This method is called after the final conditions check passes successfully. It returns an object of type `org.eclipse.ltk.core.refactoring.Change` that is used later by the refactoring user-interface to generate the change preview. We use the same change object to rewrite the classes affected by the refactoring once the programmer confirms these changes. The last step is to create an aspect that captures observer implementation to `java.util.Observer`. A proper pointcut

is created to capture where in the method that changes subject state `Observable` methods get called. These invocations get executed by the advice.

3.6 Refactoring Observer Pattern Starting with Observer

3.6.1 Refactoring Action

When the selection is one of the following cases, AJRefactor considers the type where the selection occurs one of the observers.

- The node is a method invocation where the receiver is a method parameter.
- The node is a `SingleVariableDeclaration` of a method parameter.

If the selection was evaluated to be one of the aforementioned cases, AJRefactor constructs the refactoring object. Once the refactoring object has initiated, AJRefactor would check for a number of a preconditions. If these condition checks pass successfully, the refactoring wizard pops up waiting for user input, otherwise the refactoring terminates and a message pops up indicating the reason of failure.

3.6.2 Refactoring Wizard

During the initial check, AJRefactor uses Java search engine to find the types that references the method where the selection occurs in the observer. AJRefactor considers this method observer update method. We filter the resulting methods by type. For each type if the number of methods is one, we check its body for loop statement. If it does have one, AJRefactor calculates the list of potential subjects. Then, the wizard presents these subjects to the programmer. Otherwise, the wizard involves the user in the update selection for each one of these methods.

3.6.3 Refactoring Class

Initial Conditions Checking

Aside from ensuring the project has AspectJ capability and the observer type is compiler-error free, AJRefactor checks for the following conditions. Failing to satisfy any of them results in a fatal refactoring status that immediately terminates the refactoring with a message indicating the reason of failure.

- Observer type is writable.
- There is at least one subject in the list of potential subjects or there is at least one method declaration in the list of the methods that calls the observer update. The refactoring considers

the method where programmer selection occurs the observer update method. Based on the references to this method in the entire project, the refactoring distinguishes two different cases.

  - An observer pattern implemented using the pushing technique. This is when the observer method is referenced within a loop statement. The refactoring adds the declaring type of the referencing method to the potential list of subjects.
  - An observer update case. When the observer method is not referenced within a loop statement, we consider the referencing method an instance of observer update.

Assuming the initial condition checking has passed, `checkInitialConditions` returns a `RefactoringStatus.OK` and the refactoring framework calls the `checkFinalConditions` method.

Final Conditions Checking

This method performs the following:

- *Finding Subject Changing State Methods* For each type in the subjects list, if the number of referencing methods is one and observer update is referenced within a loop statement, AJRefactor considers this the notifier method. Then, it searches for methods that calls the notifier within the subject. These methods are the methods that change the subject state and calls the notify method. The final condition checks section in 3.4 discusses the details of finding these instances.
- When finding the observer interface, observers list, and observers list handling methods, we follow the same procedure when finding the same entities for refactoring instances initiated starting with the subject class.
- *Observer Update Method* This method is the method where user selection occurs, in contrast to initiating the subject focused refactoring in section 3.4. This method get added as an inter-type declaration in the aspect. If this update is an implementation of an interface operator, we also add to the aspect type an inter-type declaration to declare the observer an implementor of this interface.
- *Changing Patterns Participants* Since this instance looks the same as the one we constructed, starting with subject class, we also register the same changes to each one of the subject classes. These changes are mentioned in 3.4
- Observer update invocations If observer update method is not referenced within a loop statement, AJRefactor presents the user with invocations found within same enclosing node i.e. the

parent node. This is to involve the user in the selection of observer update calls. When registering changes to this type of observer pattern, AJRefactor removes user selected invocations and move them into the advice created for this observer instance.

During change calculation if a `RefactoringStatus.FATAL` has returned, the programmer is presented with the unsatisfied-precondition checking-message; otherwise the method `createChange(IProgressMonitor)` is called.

Creating the Change

This method is called after the final conditions check passes successfully. It returns an object of type `org.eclipse.ltk.core.refactoring.Change` that is used later by the refactoring user-interface to generate the change preview. We use the same change object to rewrite the classes affected by the refactoring once the programmer confirms these changes. After the programmer confirms the changes, AJRefactor creates the different aspect.

3.7 Refactoring Java API Observer Pattern Starting with Observer

We consider this case a variation of the observer pattern starting with observer.

3.7.1 Refactoring Action

If AJRefactor examines user selection and finds it is one of the cases where the selected type is the observer type, AJRefactor performs one further check. AJRefactor checks if the pattern is implemented using the Java API. AJRefactor does this by checking if the observer implements `java.util.Observer` from the Java utility. If it does, AJRefactor constructs the class that handles the refactoring of the pattern implemented using Java API starting with observer type.

Once the refactoring object has initiated, AJRefactor checks for a number of a preconditions. If these conditions pass successfully, the refactoring wizard pops up waiting for user input, otherwise the refactoring terminates and a message pops up indicating the reason of failure.

3.7.2 Refactoring Wizard

Similar to the refactoring starting with observer class, this wizard displays the list of possible subjects but this list is very limited to those types that extend the `java.util.Observale`

3.7.3 Refactoring Class

Initial Conditions Checking

This method performs the same checks we perform in the refactoring instance created starting from the observer with only one difference: the list of potential subjects is calculated by finding classes that extends `java.util.Observable` class.

Final Conditions Checking

This method performs the same checks and calculations performed in the refactoring instance created starting with the subject and implemented using the common Java API classes: `java.util.Observable` and `java.util.Observer` except for the following:

- *Finding methods that change subject state* For each subject in the list of subjects selected by the programmer, we search for all methods that references the `Observable` class methods: `setChanged()`, `notifyObservers(Object arg)`, or `notifyObservers()`. This is unlike refactoring the observer implemented using the Java API where the starting type is the subject class, as we only find `Observable` method references in one method. This method is the one where the programmer selection occurs. When the refactoring starts with the subject it is known that the method where the programmer makes the selection notifies observers and has the references to `observable` methods. This permits working on a smaller set of methods which is simpler and more efficient. This is unlike the opposite case when the programmer selection is made from the observer update method as we need to know what subject this observer is watching. This is done by finding observer update referencing methods. This allows to find all subject methods that changes the subject state and notifies the observer.

- *Rewriting subjects* The changes we register are exactly the same but instead of rewriting one subject, we now register changes to all the subjects selected by the programmer.

- *Rewriting observers* These changes are also the same but in this refactoring we have only one observer which is the one where the initial selection occurs.

During change calculation if a `RefactoringStatus.FATAL` has returned, the programmer is presented with the unsatisfied-precondition checking-message; otherwise the method `createChange(IProgressMonitor)` is called.

Creating the Change

This method is called after the final conditions check passes successfully. It returns an object of type `org.eclipse.ltk.core.refactoring.Change` that is used later by the refactoring

user-interface to generate the change preview. We use the same change object to rewrite the classes affected by the refactoring once the programmer confirms these changes. The last step is to create an aspect for each method references the `Observable`. Each aspect captures observer implementation to `java.util.Observer`. Also, a proper pointcut is created to capture join points of the `Observable` methods invocations. An advice is created to execute these invocations.

3.8 Summary

This chapter discusses implementation details of AJRefactor. The implementation is handled based on the starting type and whether the pattern is implemented using the common Java API classes used to implement the observer pattern or not. Each requires user input to help in identifying pattern participants, but one case begins with the programmer selecting the observers and another with the programmer selecting the subjects. The other variation of the pattern occurs when subject change is followed by one or more method calls, whose recover is not `this`. In this case tdxhe user input involves selecting the invocations involved in the update. The pattern implementation involves how we find the different parts of the pattern, changes we make to the types participating in pattern implementation and a description of the various aspect we create as the final phase of refactoring the pattern. Appendix C explains how to get the source code of AJRefactor.

Chapter 4

AJRefactor Design

In this chapter we explain how we analyze the observer pattern components and move them into aspects. We create the various pointcuts and advice in order to capture the observed change. There are mainly two different shapes of the observer pattern that we deal with in AJRefactor.

- The first one is when the subject notifies its registered observers of a particular change in its state. This notification usually involves looping over the observers and calling their update operation. Aside from notifying the observers, subject has other methods that would allow the classes to register or unregister themselves from this observation relationship. Also, it is possible for the observer to have multiple updates based on the subject change it observers. Figure 4.1 shows an example.
- The second one is when one or more observer update methods are called in response to a subject change. These updates might be for one observer or for different observers.

in the following sections we explain using code examples how we refactor each form of the implementations we mentioned above. First we start with observer pattern implemented with the pushing technique.

4.1 Refactoring Observer Pattern Implemented with Pushing Technique

To explain this refactoring we use the example code in figure 4.1. In the example, the class `StandardDrawingView` represents the subject where the method `clearSelection()` notifies `StandardDrawingView` observers by calling the method `fireSelectionChanged()`. These observers are the `AbstractCommand` and `DrawApplet` classes. Each one of them implements the method `figureSelectionChanged()` to perform the update. This method is declared by the interface type `FigureSelectionListener` which represents the observer interface. Also, we find that `StandardDrawingView` implements the two methods `addFigureSelectionListener(FigureSelectionListener)` and `removeFigureSelectionListener(FigureSelectionListener)` that are declared in the interface type `DrawingView`. `StandardDrawingView`

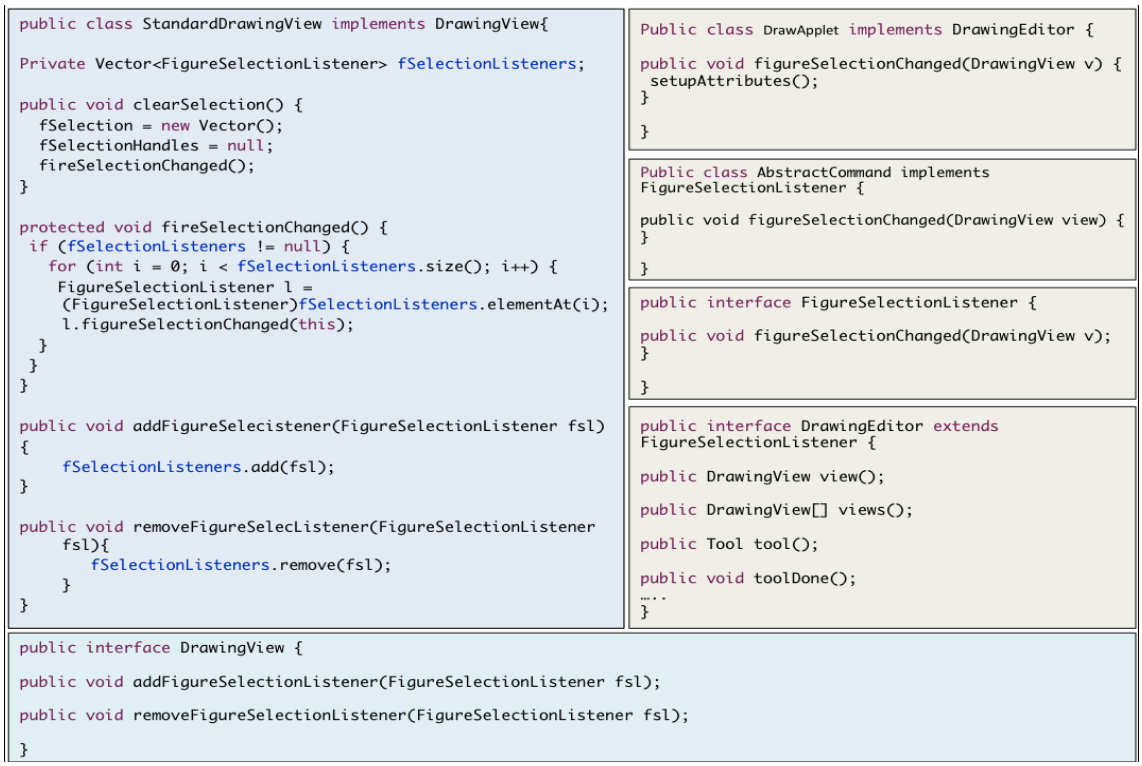

Figure 4.1: Observer Example Following GOF

declares the field fSelectionListeners of type vector which stores the observers. These are the components that AJRefactor tries to find and then create the required aspects to isolate the pattern parts into aspects.

- Once the programmer presses refactor observer button after selecting a statement from the method that changes subject state, AJRefactor examines user selection in this case (fSelectionHandles = null;). If the selection represents a change in the subject state, AJRefactor initiates the refactoring.

- AJRefactor examines the `ICompilationUnit` of the type where the refactoring was initiated. If it is free of any of errors the refactoring continues finding pattern components other wise AJRefactor terminates the refactoring presenting the programmer with a message stating that the refactoring cannot be performed in a class containing compilation errors and the programmer is requested to fix them before proceeding with the refactoring. Figure 4.2 shows an example.

- AJRefactor retrieves statements of the method that comes immediately after the selection. If their count is one and of type method invocation, AJRefactor checks its declaring class. It checks if the declaring type is the subject, any of it super classes or any of its internal classes. The method `fireSelectionChanged()` is declared by `StandardDrawingView`

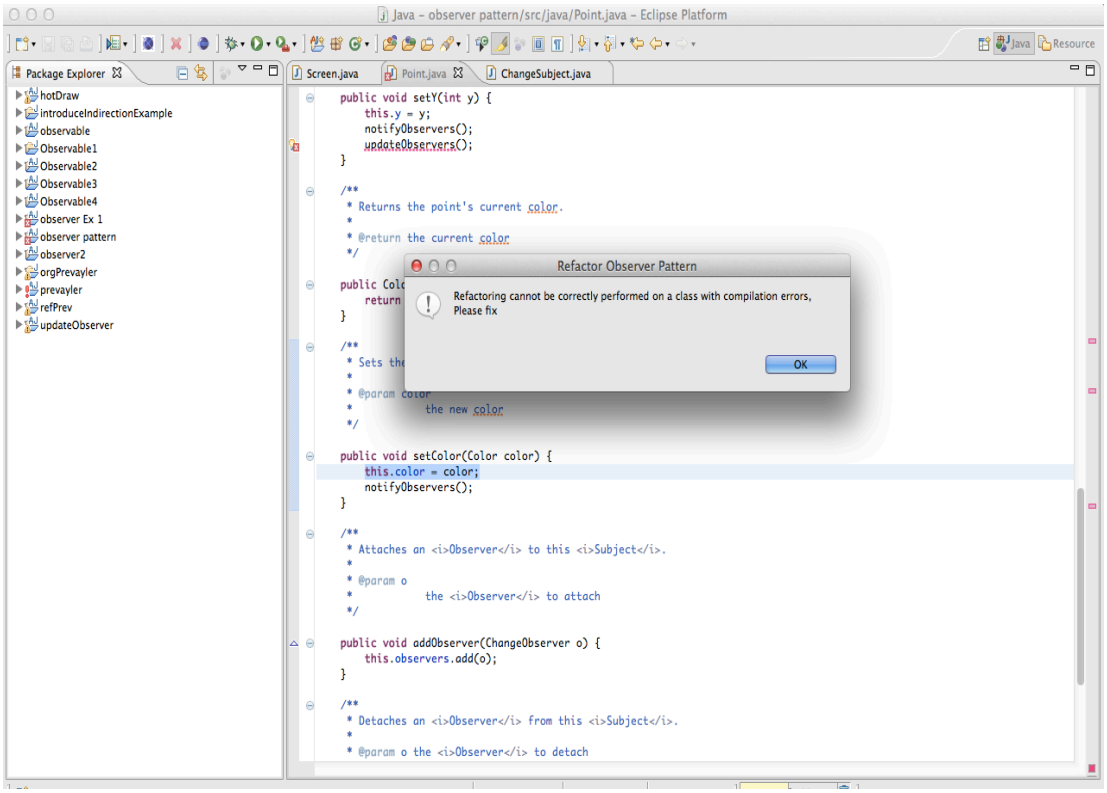

Figure 4.2: Compilation Error Message

which is the subject class. If the method declaring class is not the subject or any of its super classes or any of its local classes or if the statements count is more than one then AJRefactor considers this case is the one where the update is a number of invocations and the user has to be involved in selecting this update. For the cases where there are no statements following the statement that changes subject state or if the statement is not a method call, AJRefactor terminates the refactoring presenting the programmer with a message stating that there is no observer instance as in figure 4.3

- AJRefactor finds the potential observers from the parameters of the method `clearSelection()` which has none. Also, we add the `StandardDrawingView` field types. This is done by eliminating primitive types but adding any other type that is not an interface and not parametrized. If the field type is an interface or parametrized type whose parameter type is an interface, AJRefactor searches for its implementers and their subtypes. The process is iterative if any of the implementors is an interface type. This is particularly useful when the observer does not implement the observer interface directly but instead it implements another interface who extends the observer interface. This occurs very often.

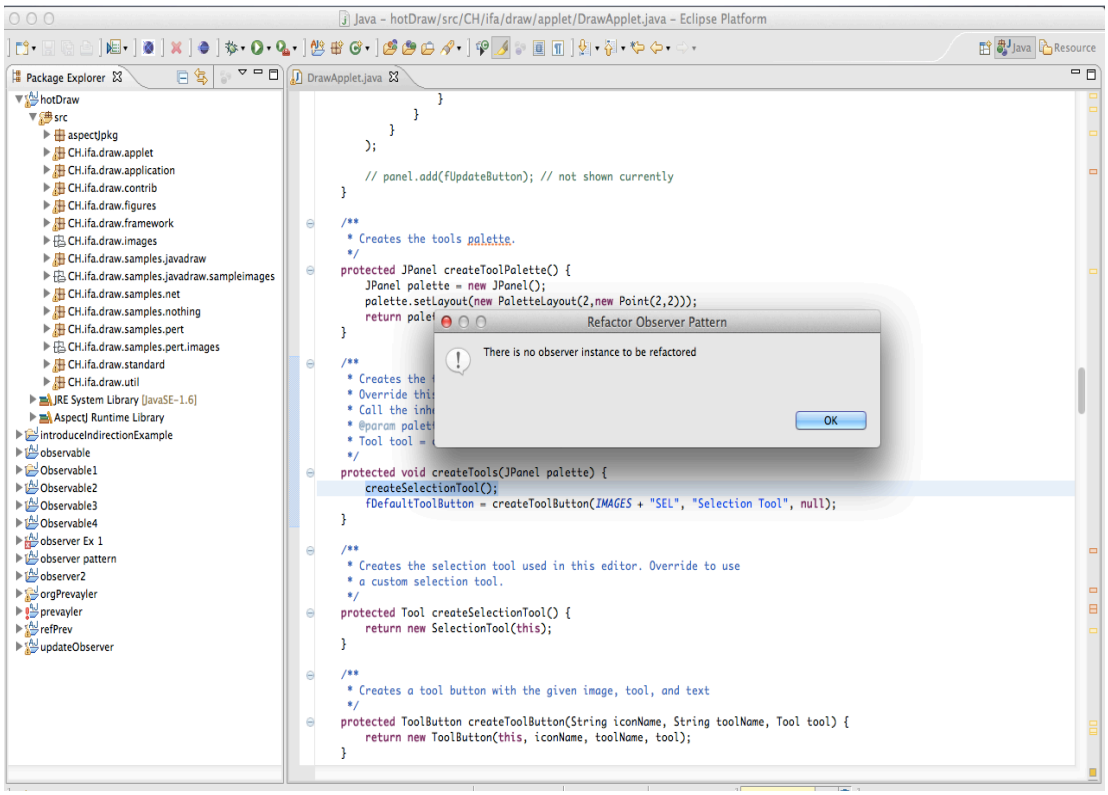

Figure 4.3: No Observer Message

- After selecting the actual observers, AJRefactor starts examining the method `fireSelectionChanged()`. Using Java searching engine, AJRefactor searches for types declaring this method. This is to find out if the method is declared by an interface type implemented by the subject or by any of its super classes. Also, AJRefactor examines the body of `fireSelectionChanged()`. In most of the cases, this method loops over the list of observers and calls their update method. Based on the loop type, AJRefactor finds the observer interface by examining the loop initializers or from its body as we explained in section 3.4.3.

- AJRefactor finds the observers list (fSelectionListeners) from the body of `fireSelectionChanged()` which is then used to find the methods that add and remove the observers. If the type of the observers list is sub-type of Java `Collection`, AJRefactor searches for the methods that call add and remove method from the collection type where the receiver is the observer list. The refactoring continues to work as intend even if these methods are not present in the subject class.

- AJRefactor finds the observer update method from the body of the loop statement. Details

on how we find them can be found in the final conditions checking section of the observer refactoring starting with subject type 3.4.

- After finding patten components, AJRefactor register the changes of `StandardDrawingView` and its observers i.e. `AbstractCommand` and `DrawApplet` to the `ASTRewrite` object. AJRefactor marks for deletion `fSelectionListeners`, `fireSelectionChanged()`, `addFigureSelectionListeners()` and `removeFigureSelectionListener()` only when they are not referenced in the `StandardDrawingView` or any of its subclasses. AJRefactor removes `FigureSelectionListener` implementation from `AbstractCommand` and `DrawApplet` when `figureSelectionChanged()` is the only method declared in `FigureSelectionListener`. Observer interface cannot be removed if it declares more than one method. When the programmer needs to refactor the other update methods, the interface implementation is needed to know the declaring types of this update method.

- When user is ready to apply the changes by clicking the ok button. AJRefactor creates three different aspects: protocol aspect where we move the components that define the subject interface into, and an instance aspect for each one of the observers. The instance aspect captures each observer and its implementation to `FigureSelectionListener`.

4.1.1 Protocol Aspect

This aspect contains the pattern components found in the subject. By analyzing the body of `fireSelectionChanged()`, AJRefactor finds the observers field, observer update method and observer interface. AJRefactor uses the method `clearSelection()` to declare the pointcut and the advice. Steps are as follow.

- We first declare a public privileged aspect.

- We move the declaration of fSelectionListeners field as an inter-type declaration of `StandardDrawingView`.

- We move the method declaration of `fireSelectionChanged` as an inter-type declaration of `StandardDrawingView`.

- We move the method declaration of the two methods that add and remove observers as an inter-type declaration of `StandardDrawingView`.

- The add and remove methods are primarily an interface operations declared in DrawingView. Beside these two methods, `DrawingView` has many other operations therefore we remove them from `DrawingView` and declare an interface `IDrawingView` in the protocol aspect with these two operations. Finally, we declare `DrawingView` to extend this interface. This

```
public privileged aspect StandardDrawingViewProtocol{

    public interface IDrawingView {
        public void addFigureSelectionListener(FigureSelectionListener fsl);
        public void removeFigureSelectionListener(FigureSelectionListener fsl);
    }

    declare parents: DrawingView extends IDrawingView;

    private  Vector<FigureSelectionListener> StandardDrawingView.fSelectionListeners =
        new Vector<FigureSelectionListener>();

    public void StandardDrawingView.addFigureSelectionListener(FigureSelectionListener
        fsl) {
        fSelectionListeners.add(fsl);
    }

    public void StandardDrawingView.removeFigureSelectionListener(FigureSelectionListener
         fsl) {
        fSelectionListeners.remove(fsl);
    }

    private void StandardDrawingView.fireSelectionChanged(){
        if (fSelectionListeners != null) {
            for (int i = 0; i < fSelectionListeners.size(); i++) {
                FigureSelectionListener l = (FigureSelectionListener)fSelectionListeners.
                    elementAt(i);
                l.figureSelectionChanged(this);
            }
        }
    }
}
```

Listing 4.1: StandardDrawingView Protocol Aspect

would prevent breaking the hierarchy of the classes that implements the `DrawingView`. This operation is currently not supported in AJRefactor and we had to do it manually. That is after creating the protocol and the instance aspects, we manually add a declaration of a new interface that holds the method declarations of the methods that adds to and removes from the list of observers. This is for the cases where the subject interface is not declared as a separate type. This makes it difficult to isolate the subject interface.

Now the protocol aspect is considered complete but the pattern refactoring is not. We must declare a separate aspect for each observer.

4.1.2 Observer Update Instance Aspect

For each observer, this aspect captures the observer interface implementation. It also declares the pointcut that captures the call to `fireSelectionChanged()` based on its location in `clearSelection()`. Returning to the example, part of changing `AbstractCommand` and `DrawApplet` involves moving the implementation of `FigureSelectionListener` into the instance aspect. Listings 4.5 and 4.4 are the instance aspects for the `AbstractCommand` and `DrawApplet` respectively.

- We first declare a public privileged aspect because we require access to private members used

```
1  public privileged aspect DrawAppletOfClearSelection{
3      declare parents: DrawingEditor extends FigureSelectionListener;
5      public pointcut clearSelectionPointcut(StandardDrawingView sdw):
        call(void clearSelection()) &&
7       target(sdw);
9      after(StandardDrawingView sdw): clearSelectionPointcut(sdw){
           sdw.fireSelectionChanged();
11     }
13     public void DrawApplet.figureSelectionChanged(DrawingView view) {
           setupAttributes();
15     }
   }
```

Listing 4.4: DrawApplet Instance aspect

within the body of the observer update method.

- Declaring observer inter-type declaration depends on the observer itself. The observer can have one of three cases:
  - It can be a direct implementor of the observer interface `FigureSelectionListener` such as `AbstractCommand`. In the instance aspect of `AbstractCommand` we declare:

```
1      declare parents: AbstractCommand implements FigureSelectionListener;
```

Listing 4.2: AbstractCommand Intertype decalration

  - It might implement an interface that extends the observer interface such as `DrawApplet`. `DrawApplet` implements `DrawingEditor` which extends `FigureSelectionListener`. In this case we remove extends `FigureSelectionListener` from the `DrawingEditor` and replace it with the following inter-type declaration.

```
1      declare parents: DrawingEditor extends FigureSelectionListener;
```

Listing 4.3: DrawApplet Intertype declaration

  - It might be a sub-type of any of the previous two types. In this case we leave the type as it is because it will be handled when its super type is handled.
- We add the inter-type declaration of the `figureSelectionChanged()`.
- We define the pointcut and the relative advice.

This refactoring helps isolating the pattern into the aspect protocol which is now entirely responsible for the whole notification while each one of the instance aspects now serves the role of

```
public privileged aspect StandardDrawingViewAspectOfAbstractCommand {

    declare parents: AbstractCommand implements FigureSelectionListener;

    public pointcut clearSelectionPointcut(StandardDrawingView sdw):
        call(void clearSelection()) &&
        target(sdw);

    after(StandardDrawingView sdw): clearSelectionPointcut(sdw){
        sdw.fireSelectionChanged();
    }

    public void AbstractCommand.figureSelectionChanged(DrawingView view) {
    }
}
```

Listing 4.5: AbstractCommand Instance Aspect

updating each observer. These observers need not to worry about this update as it is now handled by the aspect.

4.1.3 Generating Pointcut and Advice

We divide the general shape of methods that notifies a subject observers into three different shapes bases on the parenting node of the statement that represent the subject change and the statement that represent the call to the notify method. Since the abstract syntax tree is a tree of nodes where some nodes might have children nodes, Eclipse allows the programmer to query each node for its children and for its parent nodes. The top node is known as the parent node. For example, the call `fireSelectionChanged()` is a child node to the `Block` that represent the method body of `clearSelection()` method in figure 4.4. This means we can say that the `Block` is the parent node of the `fireSelectionChanged()` invocation. Because the `Block` node itself is a child node of `clearSelection()` method declaration, this `MethodDeclaration` is also an indirect parent to the call `fireSelectionChanged()`. Below we describe these three shapes.

- In this shape, both statements, the one that changes the state of the subject and the notify method invocation (e.g. `fireSelectionChanged()`) are parented by the method declaration. The method `clearSelection()` in figure 4.4 is an example. To refer to this shape later we call it (MethodDecSubjectChange - NotifyCall).

- In this shape both subject change and notify call are parented by an if statement. Subject change is the if expression while the method invocation to notify is located in the then part of the if. The method `removeFromSelection()` in figure 4.4 shows an example. To refer to this shape later we call it (IfExpSubjectChange - NotifyCall).

- In this shape both subject change and notify call are parented by the then statement of an if. The method `addToSelection()` in figure 4.4 shows an example. To refer to this shape

later we call it (ThenStmntSubjectChange - NotifyCall). For cases where both statements are part of an else statement, we manually reverse the condition for AJRefactor to be able to refactor it.

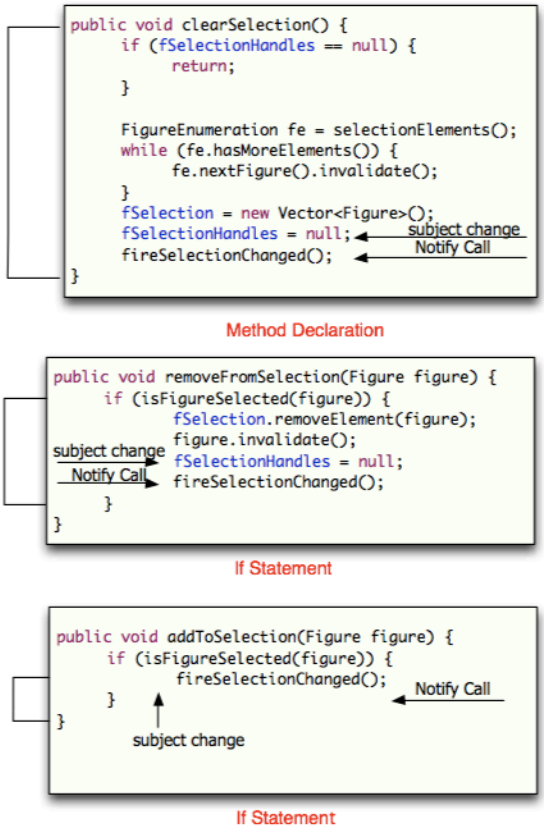

Figure 4.4: Parent of the subject change and notify method

Below we explain how AJRefactor creates the pointcut and advice for each shape. We also explain the changes we had to make before refactoring in order for AJRefactor to work.

MethodDecSubjectChange - NotifyCall

The method `toggleSelection` in figure 4.5 shows an example of this shape. The body of this method has an if statement before calling `fireSelectionChanged()`. Before we can use AJRefactor, we used extract method Java refactoring to extract the if statement into a method. The new method call will be the subject change we start refactoring with. Method invocation to `fireSelectionChanged()` comes at the end of the toggleSelection body, therefore, AJRefactor creates an after advice that executes when toggleSelection get called by an object of type `StandardDrawingView`. Before adding the pointcut and advice to the aspect body, AJRefactor creates each one of them as a string. Examining the call to `fireSelectionChanged()`, AJRefactor adds an object of type `StandardDrawingView` to the pointcut arguments as it is the

target object when `toggleSelection()` get called. This object is also attached as the receiver of `fireSelectionChanged()` because its expression is void. For other expression cases we perform the following analysis.

- If the expression is `this`, AJRefactor replaces this with the target object.
- If the expression is a simple name of a subject field, AJRefactor attaches the object type `StandardDrawingView` to the method call.
- If the expression is a simple name of a method argument, AJRefactor adds this argument to the pointcut declaration and to the `args` construct of the pointcut declaration.
- If the expression is a simple name of a locally-declared variable. We manually change this variable by inlining the value that was originally assigned to it before applying AJRefactor. This value can be one of the following cases:
  - a field,
  - a method argument,
  - a method call where the receiver is a field,
  - a method call where the receiver is *this* or
  - a method call where the receiver is a method argument.
  - a method call where the receiver is the `super` expression.

IfExpSubjectChange - NotifyCall

The method in figure 4.6 is an example of this shape. Before refactoring, we extracted the value of the two local variables into method declarations then we used them in the if expression. The call to `getEventDispatcher().fireCommandExecutableEvent()` is the only statement in then statement body. AJRefactor examines this call and the if expression to calculate pointcut parameters. The expression in the `if` statement is an infix expression with an && operator. The expression of each method invocation is the null expression; AJRefactor adds `AbstractCommand` to the pointcut parameters. Also, each call uses an object of type `DrawingView` (oldView and newView) which also get added to the pointcut parameters list. The final pointcut is the execution of `viewSelectionChanged()` when the if expression evaluates to true where the arguments are oldView and newView and the target object is of type `AbstractCommand`. After this pointcut reaches its join point, the call `getEventDispatcher().fireCommandExecutableEvent()` get executed. Figure 4.6 shows `viewSelectionChanged` before and after refactoring.

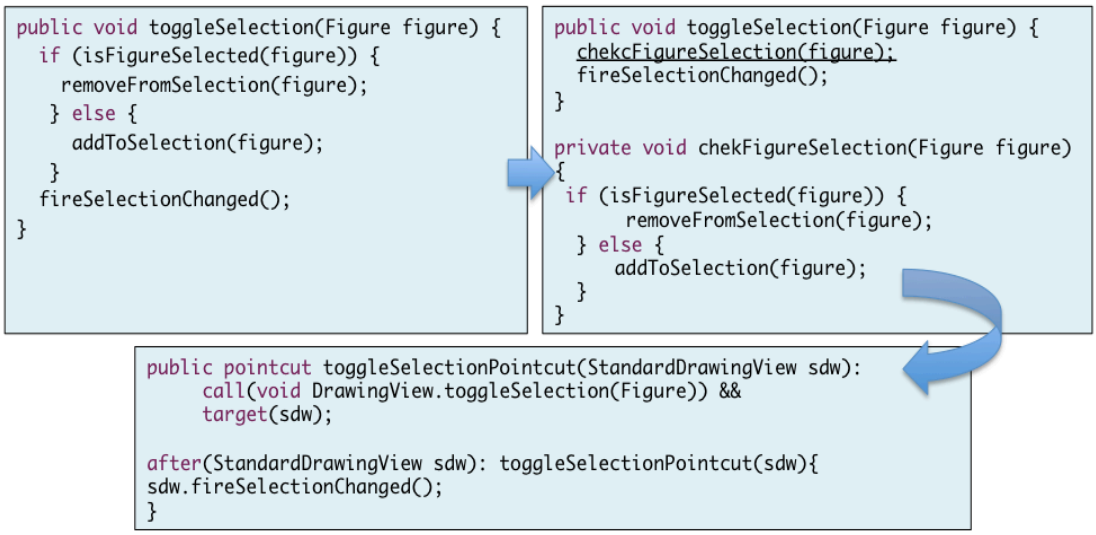

Figure 4.5: MethodDecSubjectChange - NoifyCall

ThenStmntSubjectChange - NotifyCall

The method `removeFromSelection` in figure 4.7 is an example of this shape. The statement `figure.invalidate()` does not change a subject state. We want to advise the call to `fireSelectioChanged()`. Yet there is nothing wrong with statements before this call. Selecting the assignment statement (`fSelectionHandles = null;`) will cause AJRefactor to consider this case as there are multiple updates and user involvement is needed to select them. That would work except that we want to refactor this instance as one that takes the first from of the observer pattern we handle with AJRefactor (i.e. implemented with the pushing technique), therefore, we make a small change to make it work as intended. We exchange between (`fSelectionHandles = null`) and (`figure.invalidate()`). Now, we can select (`fSelectionHandles = null`) as the change statement and advice the call `fireSelectioChanged()` with an after advice. Since this call is within an if statement, AJRefactor creates an if pointcut when `removeFromSelection` get executed by an object of type `StandardDrawingView`. AJRefactor creates this pointcut with one parameter of type `Figure`. Figure object is used in the expression of if statement. Figure 4.7 shows the refactoring of the method `removeFromSelection`.

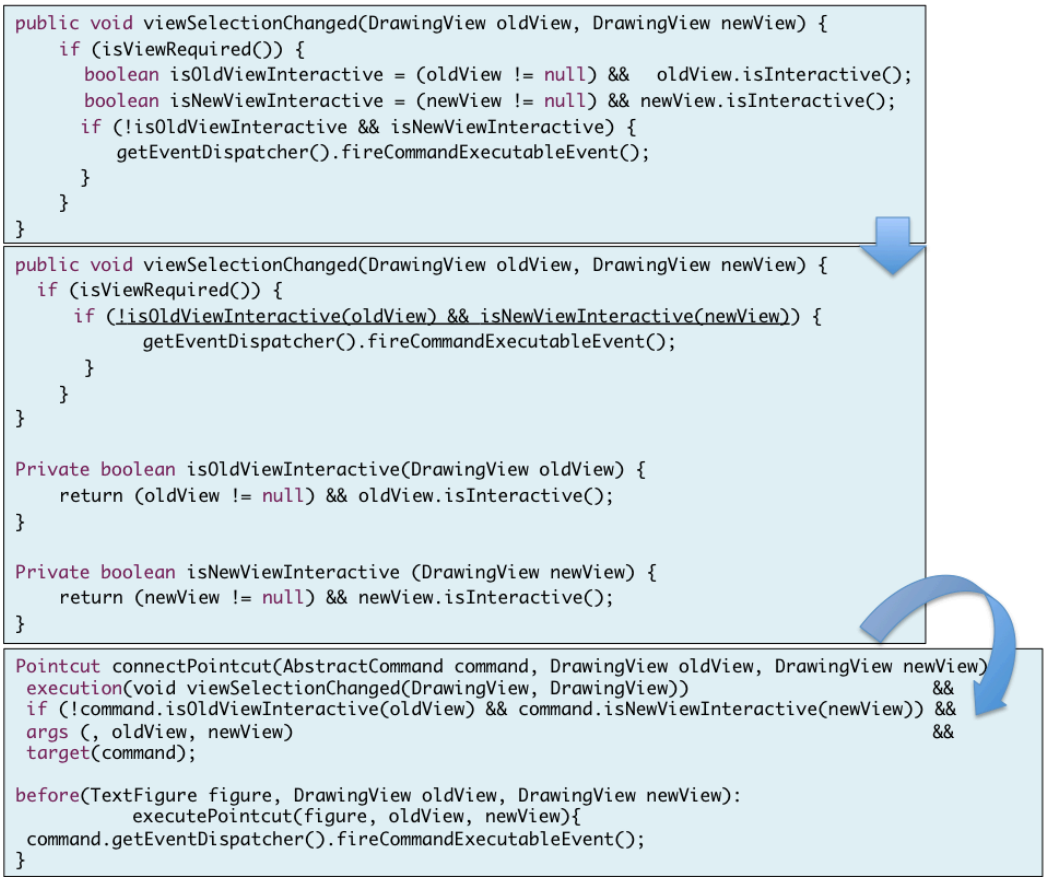

Figure 4.6: IfExpSubjectChange - NotifyCall

4.1.4 Limitations

There are some limitations when refactoring this form of the observer pattern. These limitations are

- One of the sources that AJRefactor calculates the potential observers from is the list of fields. When the type of the list of observers field is of parametrized type and the parameter type is not provided then AJRefactor cannot find these observers and list them when the wizard pops up. Therefore, we had to manually add the parameter type of the parametrized type before starting the refactoring with AJRefactor so we can get these observers.

- AJRefactor does not refactor the calls to the methods that adds and removes the observers from the observers list as part of the refactoring process. Yet these can still be refactored using AJRefactor as an observer update call as long as they are parented by an if statement or by a method declaration and if not these must be refactored manually.

- AJRefactor implementation uses the Java searching engine in different steps of the refactoring process. For example, finding implementors of a type, finding types declaring a method and

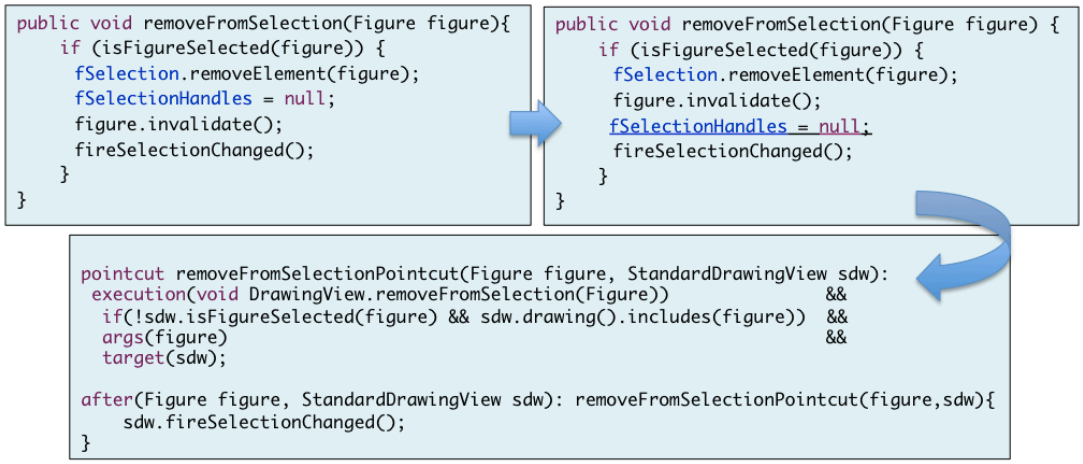

Figure 4.7: ThenStmntSubjectChange - NotifyCall

finding types referencing a method. Although we used this engine as described in Java API documentation, there are some cases where this engine does not give all the desired results.

- In the example aspect in figure 4.4, `clearSelection` method calls `fireSelectionChanged`. Another method (`toggleSelection` for example) also calls `fireSelectionChanged`, AJRefactor would create an aspect for each one of these methods for each observer. This means when any of these two methods get executed, the method `figureSelectionChanged` get called. Although valid, AJRefactor, creates two different aspects for each method. Each aspect has the inter-type declaration of the method `figureSelectionChanged`. This causes the AspectJ compiler to complain because there are two declarations of the same method. One possible solution for this is to create a pointcut for each one of the subjects' methods in one aspect along with the observer update method. Figure 4.6 shows an example. In its current version, AJRefactor design does not handle this situation.

Below we explain with examples the refactoring of observer updating calls or the calls to remove and add observers.

4.2 Refactoring Update Calls

This is the second form of observer pattern we tackle in AJRefactor. When the number of statements after the selection is more than one or it is one statement of type method call whose declaring type is not the subject or any of its super types or local types. We display these statements to the programmer so she can select the one that updates observers. This implementation of observer pattern also takes three different shapes. These shapes are

```
1  public privileged aspect StandardDrawingViewAspectOfDrawApplication{
2
3    declare parents:DrawingEditor extends FigureSelectionListener;
4
5    public pointcut clearSelectionPointcut(StandardDrawingView sdw):
6     call(void clearSelection()) &&
7     target(sdw);
8
9    after(StandardDrawingView sdw): clearSelectionPointcut(sdw){
10       sdw.fireSelectionChanged();
11     }
12
13   public pointcut fireSelectionChangedPointcut(StandardDrawingView sdw):
14     call(void DrawingView.toggleSelection(Figure)) &&
15     target(sdw);
16
17  public pointcut removeFromSelectionPointcut(Figure figure, StandardDrawingView sdw):
18   execution(void StandardDrawingView.removeFromSelection(Figure))  &&
19   if (sdw.isFigureSelected(figure))                                 &&
20   args(figure)                                                      &&
21   target(sdw);
22
23  after(Figure figure, StandardDrawingView sdw): removeFromSelectionPointcut(figure,sdw){
24      sdw.fireSelectionChanged();
25    }
26
27  pointcut addToSelectionPointcut(Figure figure, StandardDrawingView sdw):
28    execution(void StandardDrawingView.addToSelection(Figure))          &&
29    if(!sdw.isFigureSelected(figure) && sdw.drawing().includes(figure)) &&
30    args(figure)                                                        &&
31    target(sdw);
32
33  after(Figure figure, StandardDrawingView sdw): addToSelectionPointcut(figure,sdw){
34       sdw.fireSelectionChanged();
35   }
36
37  public void DrawApplet.figureSelectionChanged(DrawingView view) {
38       setupAttributes();
39  }
40 }
```

Listing 4.6: DrawApplet Complete Instance Aspect

- In this shape the subject change and observers update are parented by the method declaration. We call this MethodDecSubjectChange - multiInvocations.
- In this shape the subject change and observers update are parented by an if statement. If expression is the subject change while observers update is in the then statement. We call this IfExpSubjectChange - multiInvocations.
- In this shape the subject change and observers update are parented by an if statement. Both subject change and observers update is in the then statement of an if. We call this ThenStmntSubjectChange - multiInvocations. For the case when both the subject change statement and the notify call are both parented by an `ElseStatement`, we reverse the condition before we can apply AJRefactor as it does not handle the `ElseStatement` case.

4.2.1 MethodDecSubjectChange - multiInvocations

The method `connect` in figure 4.8 represents an example of this shape. The method `connect` calls `addFigureChangeListener(FigureChangeListener)` on an `fObservedFigure` object. This call is located just before the call to `updateLocation()` declared by `TextFigure`, therefore, AJRefactor creates a pointcut to capture the call to `updateLocation()` of type `TextFigure` when executed within code of `connect` of type `TextFigure`. The expression (`fObservedFigure`) of the call `fObservedFigure.addFigureChangeListener(this)` is a field declared by `TextFigure`. AJRefactor adds an `Object` of type `TextFigure` to pointcut parameters. A before advice is created to execute the call `fObservedFigure.addFigureChangeListener(this)` when the pointcut reaches its join point. Figure 4.8 shows the method `connect` before and after refactoring.

4.2.2 IfExpSubjectChange - multiInvocations

Figure 4.9 shows an example of this shape. The method `change` calls `figureRequestUpdate(new FigureChangeEvent(this))` when if's expression evaluates to true. Therefore, AJRefactor creates an execution pointcut for the method `change` with an if join point. AJRefactor adds an object of type `GraphicalCompositeFigure` to parameters of the pointcut. The method call `figureRequestUpdate(new FigureChangeEvent(this))` is the last statement in then statement, therefore, we declare an after advice to execute when the pointcut reaches its join point. Figure 4.9 shows the method `change` before and after refactoring.

4.2.3 ThenStmntSubjectChange - multiInvocations

Figure 4.10 shows an example of this shape. The two statements `clearSelection()` and `fDrawing.removeDrawingChangeListener(this)` are part of then statement of the first if

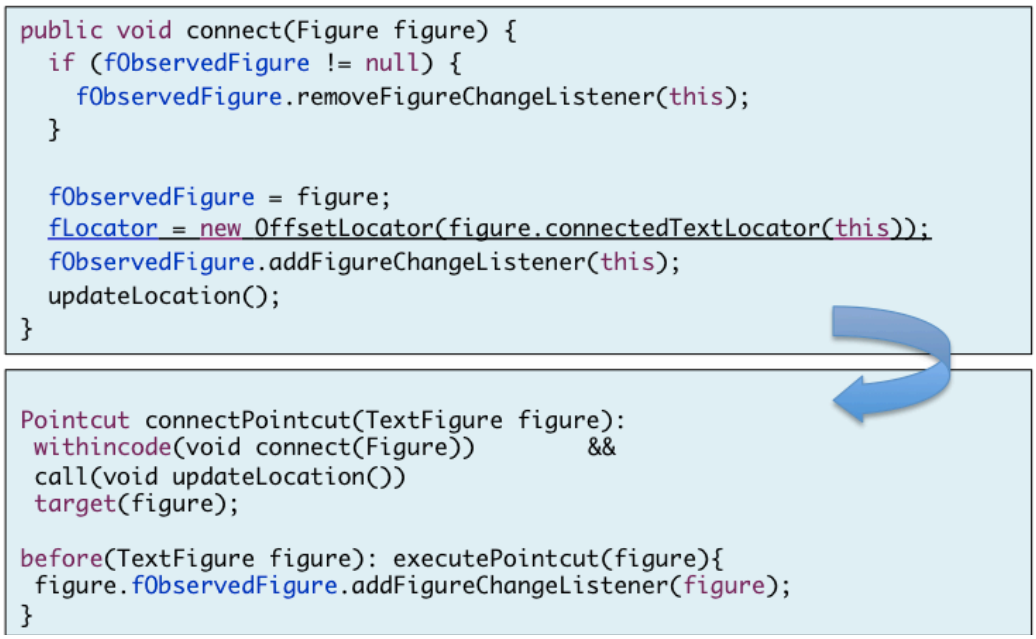

Figure 4.8: Observer Update: MethodDecSubjectChange - multiInvocations

statement in the method 4.10. We want to execute the invocation to `fDrawing.removeDrawingChangeListener(this)` which is the last statement of then statement. AJRefactor adds only one object of type `StandardDrawingView` to the pointcut parameters. We use this object to replace *this* expression. Also it is used to access the call to remove as its expression is a field declared by `StandardDrawingView`. AJRefactor defines an after advice that get executed when the method `setDrawing` executes on an object of type `StandardDrawingView` and `fDrawing` does not equal to null. Figure 4.10 shows the method `setDrawing` before and after refactoring.

4.2.4 Limitations

When refactoring the observer update calls there are some cases that AJRefactor could not refactor due to the following reasons.

- During the refactoring we found cases where the observer update calls were parented by the try/catch block or by the synchronize block. Current version of AJRefactor does not handle these situations.

- There are cases when the update calls or even the call to the method that notifies observers uses a method-local variable. Since AJRefactor does not analyze these cases, we try to make these situations work as possible. We manually change the code before applying AJRefactor. This is done by inlining the local variable with the value that was assigned to it.

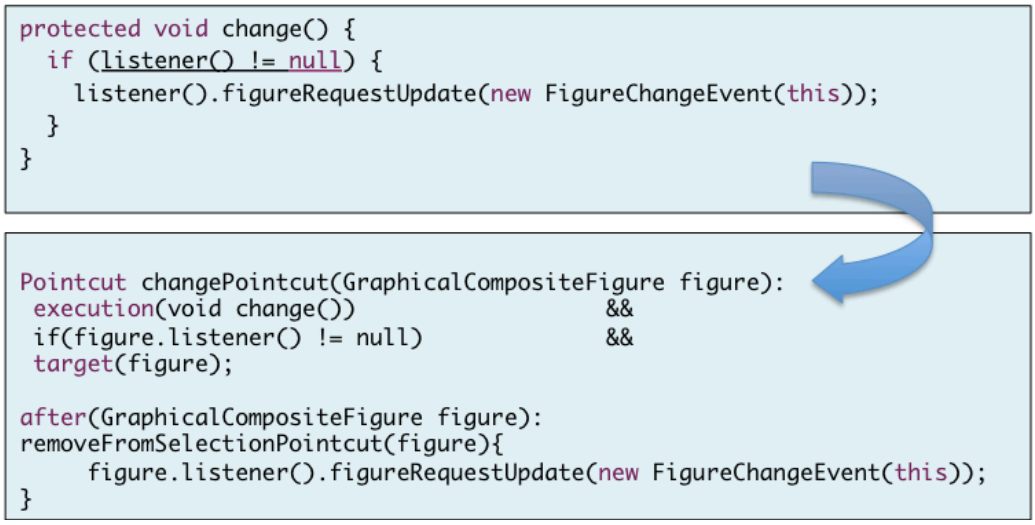

Figure 4.9: Observer Update: IfExpSubjectChange - multiInvocations

4.3 Summary

This chapter explains the different heuristics that AJRefactor can refactor and how AJRefactor refactors each one of them. In general AJRefactor handles tow different shapes of the observer pattern. One where the observer is implemented with the pushing technique. The other one is when observers watches for a change in the subject state and then call their update method. When creating the advice and the pointcut we divide the method that changes subject state into three different forms based on the parenting node. Table 4.1 summarizes these shapes. We also point out the tool limitations and their different sources. The source of these limitations can be one of the following:

- AJRefactor implementation.
- The way the source code is written and shaped.
- Java API, though it is very limited.

In the next chapter we present our experiment results and evaluation.

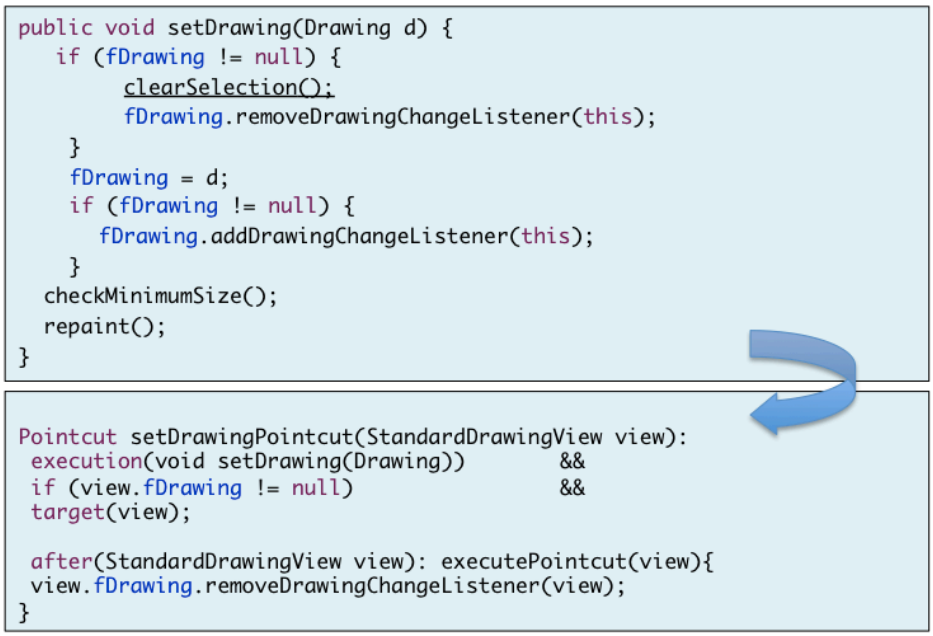

Figure 4.10: Observer Update: ThenStmntSubjectChange - multiInvocations

Table 4.1: Shape of the Method Changing Subject State by Observer Pattern

Observer Pattern	Shape
Implemented with pushing technique	MethodDecSubjectChange - NotifyCal
	IfExpSubjectChange NotifyCall
	ThenStmntSubjectChange - NotifyCall
Observer Update	MethodDecSubjectChange - multiInvocations
	IfExpSubjectChange - multiInvocations
	ThenStmntSubjectChange - multiInvocations

Chapter 5

Results and Evaluation

Using case studies we tested AJRefactor on two well-known Java projects: JHotDraw [2] and Prevayler [3].

JHotDraw is an open source graphical editor. It has been developed with a heavy use of design patterns including the observer pattern. There are many refactoring projects that used JHotDraw as a bench mark to perform manual and automatic refactorings. Although there is newer version of this project, we used 5.3 the last packaged version. This version of JHotDraw was developed with a total of 398 classes and a total of 40,022 lines of code. JHotDraw was manually refactored to AspectJ. The refactored version is publicly available for download from [6]. The refactoring experiment is completely described in [27]. We have used the refactored version as a guide to find the observer pattern instances and refactor them using our tool.

Prevayler is an open source object persistence library for Java. Prevayler is part of Brekely Software Distribution (BSD) which is a Unix operating system. We used version 2.5 of Prevayler. This version was developed with a total of 53 classes and 1745 lines of code. Prevayler is distributed through Apache Maven. We found some sites supporting developers who use Prevayler library.

We selected these two projects in order to test AJRefactor on Java applications with varying size. JHotDraw is a medium size project; while Prevalyer is a small one. Furthermore, JHotDraw is well studied**??**, and well-accepted decompositions into design patterns are available. This include the observer pattern. This gives us confidence on the coverage and correctness of AJRefactor. Prevayler is intended to provide another real-world application, and give metrics for novel systems.

We report our refactoring results based on different criteria to show the value of refactoring the observer pattern with AJRefactor.

5.1 Results by Instance Shape

In this section we evaluate AJRefactor functionality by reporting the cases that tool was able to refactor and also the ones could not be refactored. The discussion also involves describing the reasons behind the tool being unable to refactor some of these cases. We divide the discussion into two parts based on the main shape of the pattern instance.

5.1.1 Observer Pattern Implemented with Pushing Technique

In both projects, we found 70 instances implemented using the pushing technique(66 in JHotDraw and four in Prevayler). The 66 cases where previously identified by others[27]. Sixty of the 70 instances were successfully refactored by AJRefactor and 10 of them failed to be refactored. We found two (from JHotDraw) of these 10 could not be refactored because Eclipse Java search engine could not retrieve the observer update methods. We depend on Eclipse Java searching engine to find several entities of the pattern. Although we use this engine as described in the documentation, there are a few situations where this engine does not provide all the expected results. We have encountered this with two observers: `CH.ifa.draw.standard.AbstractCommand` and `CH.ifa.draw.standard.AbstractTool` that observe the change in the view of the `CH.ifa.draw.application.DrawApplication` class. For these two observers, the engine could not retrieve the observer's update method. The other eight instances could not be refactored by AJRefactor for several reasons. These include the following:

- the instance does not take a shape that can be refactored with AJRefactor. For example, the parenting node of the notify method is a try/catch block.
- there was no subject change that can be selected to trigger the refactoring.

In total, AJRefactor was able to refactor 60 of these instances which represents 85% of the total cases. This percentage is respectable. This is because this shape is the first shape we considered when we first started implementing the tool. Hence, we were able to analyze the different cases this shape might take. Also, we were able to analyze the different implementations the notify method could take. Overall, this shape is well covered by AJRefactor. Refactoring this shape gives the most benefit as it completely isolates the observer pattern into the different aspects. It also frees the observers from needing to update themselves and from the observer interface implementation. Table 5.1 shows the refactoring results of this type of observer pattern. Results are categorized by instance shape.

For detailed information about each of these instances, please refer to appendix A.

5.1.2 Refactoring Update Calls

The results we report in table 5.2 includes both the call of the update methods and the call to the methods that adds and removes the observers. We found in total 45 instances of which 28 instances were successfully refactored with AJRefactor. There were 17 instances which could not be refactored with AJRefactor for several reasons. These include

- the method does not change a subject field, or

Table 5.1: Results of refactoring the observer pattern - The pushing technique

Shape	Total Found	Refactored by AJRefactor
IfExpSubjectChange - NotifyCall	36	36
ThenStmntSubjectChange - NotifyCall	16	14
MethodDecSubjectChange - NotifyCall	10	10
Others	8	0
Total Instances	70	60
%	85	

Table 5.2: Update Calls Refactoring Results

Shape	Total Found	Refactored by AJRefactor
IfExpSubjectChange - multiInvocations	14	14
ThenStmntSubjectChange - multiInvocations	1	1
MethodDecSubjectChange - multiInvocations	13	13
Others	17	0
Total Instances	45	28
%	62	

- the update method is called from the constructor which is not handled by AJRefactor, or
- the instance does not take one of the shapes we explained in chapter 4.

In total, AJRefactor was able to refactor 62% of the cases.

Adding support for these shapes currently not supported by AJRefactor is not hard. It only requires implementing a visitor to visit these nodes (e.g. try/catch block node and synchronization block node) and then finding the pattern instance within these node types. Based on the refactoring experience of JHotDraw and Prevayler, there was no observer instances within a try/catch block or within synchronized block in JHotDraw while in Prevayler there was two instances within synchronized block ad one instance within try/catch block.

5.2 LOC Assessment

There are two different metrics to measure a program's of code: SLOC and LOC. The SLOC measures the source lines of code including blank and comment lines. However, the LOC measures the program's source lines that represents an executable statement in any particular language. Blank and comment lines are excluded. We used the Metric tool provided by the CodePro Analytix Eclipse plug-in [4]. LOC metric in CodePro only exclude comments and blank lines but everything

```
package org.prevayler;

 /**
 * Tells the time.
 */
public interface Clock {

    /** Tells the time.
     * @return A Date greater or equal to the one returned by the last call to this
         method. If the time is the same as the last call, the SAME Date object is
         returned rather than a new, equal one.
     */
    public java.util.Date time();

}
```

Listing 5.1: LOC Metric in CodePro Analytix plug-in

else is counted as part of the code including lines with single or double curly brackets. For example, the LOC metric for the `Clock` class in listing 5.1 is four.

This section evaluates the code size before and after refactoring. The question which rises is whether refactoring to aspects reduces Java code size or not. In most of the cases, observer pattern implementation involves many classes. Aspects encapsulate pattern entities by moving them from their original classes into aspects. This suggests the code size might decrease. To answer this question we discuss the numbers in table 5.3. The table shows the total number of lines, fields and methods of JHotDraw and Pevayler before and after the refactoring. These numbers clearly shows a small reduction in the total number of lines for both projects. The code size of JHotDraw has reduced by 1.3%, while Prevayler code size decreased by 1.8%.

After the refactoring the number of classes has reduced by 3% in JHotDraw while the classes remained unchanged in Prevayler. Classes are only deleted when there are no direct references to them or to any of their methods. As discussed later, reflective access is not be recognized, so this is a place for potential improvement.

The number of methods has reduced by 5% for JHotDraw while they remain the same for Prevayler. After the refactoring we move all subject methods that notify, add and remove observers and observers' update methods to be part of the aspect code. Now, the classes are oblivious to the observation relationship and are more focus on their original task. Overall, we see that aspect-oriented refactoring has decreased the code size by a small percentage.

Table 5.3 also shows the code size including aspects. These results are reported by the column *total* for both projects. Similar to any application, the LOC metric does not include library classes included in the project class path, and so the total excludes the abstract aspects. The reason is these aspects are now considered a library aspects that can be referenced and extended by any aspect to be part of the observer relationship represented by that abstract aspect. The *aspect* column represents the different metrics for the created aspects rustling from the refactoring. It

is clear that LOC with aspects has increased as well as the number of methods. However, the complexity of the existing code is improved, as the DSM analysis shows below.

Table 5.3: JHotDraw and Prevayler Metrics

Application	JHotDraw				Prevayler			
Metric	Before	After	Aspect	Total	Before	After	Aspects	Total
LOC	14,611	14,425	1,245	15,670	1,745	1,713	108	1821
Classes/Aspects	273	266	26	292	53	53	6	59
Fields	489	471	0	471	107	107	0	107
Methods	2,034	1,942	76	2018	199	199	1	200

5.3 Modularity Assessment

To assess code modularity of the classes before and after the refactoring, we analyze the Dependency Structure Matrix (DSM). We used The analyze Dependency Matrix tool from the IntelliJ IDEA integrated development environment [7]. Below we evaluate code modularity for JHotDraw and Prevayler.

5.3.1 JHotDraw

Figure 5.1 is the DSM of JHotDraw in its original version. The number in each cell represents the column dependency on the row. Types presented are those involved in the pattern refactoring. There is a tight coupling between different classes of the project. Part of this dependency is related to the observer pattern. After refactoring the observer pattern, DSM now looks like the one shown in 5.2. The black boxes and the red circles in both figures highlights the most notable dependencies related to the different observer instances and how they change after the refactoring.

After aspect-refactoring modularization of the pieces of code related to the observer pattern, there is a dramatic change in the DSM. This is specially true for the types from the framework package. Before refactoring the pattern there was a tight coupling between these types but after refactoring this coupling has loosened. Some of these types represent the different interfaces that observers implement to be able to update themselves when their target subject has changed its state.

Before refactoring into aspects, there is a great dependency on the type `FigureChangeListener` but after the refactoring this coupling has completely eliminated except for the following

- the dependency of the type `FigureChangeEventMulticaster` on `FigureChangeListener` has reduce by one

- the type `AbstractFigure` still has two dependencies on `FigureChangeListener`

The coupling between the observer classes and their observer interface `ViewChangeListener` has completely eliminated except for the type `JavaDrawViewer` it remained unchanged.

There is a tight coupling between the classes `DrawingView`, `StandardDrawingView`, and `StandardDrawing` and `DrawingChangeListener` before the refactoring. After the refactoring this coupling has been almost eliminated. The same can be said about the coupling between the classes `UndoableCommand`, `AbstractCommand`, and `CommandMenu` on their observer interface `CommandListener`. This coupling has almost eliminated.

Also, there is a great dependency on `DrawingEditor` and `DrawingView` before the refactoring but after the refactoring into aspects many of these dependencies has loosened. Another notable tight coupling can be seen between the types `ToolButton`, `AbstractTool`, and `UndoableTool` on their observer interface `ToolListener`. After the refactoring this coupling has almost eliminated. As a result, the classes no longer depend on the other classes that implements the observer pattern.

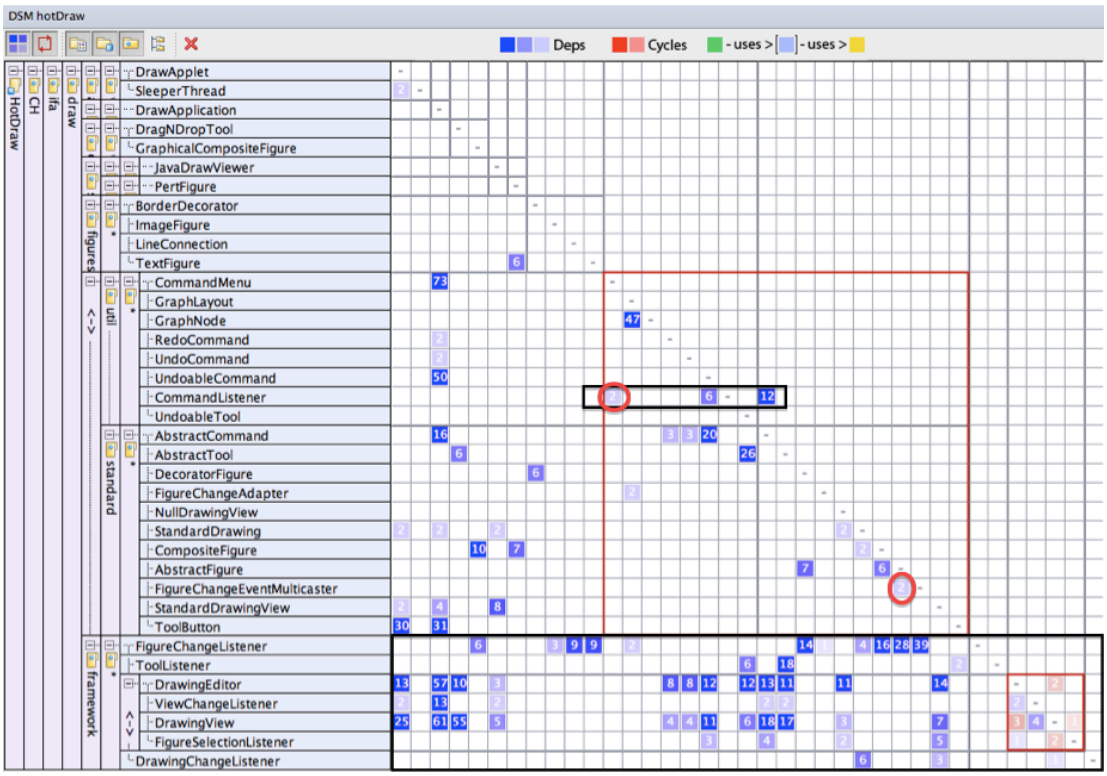

Figure 5.1: DSM - JHotDraw

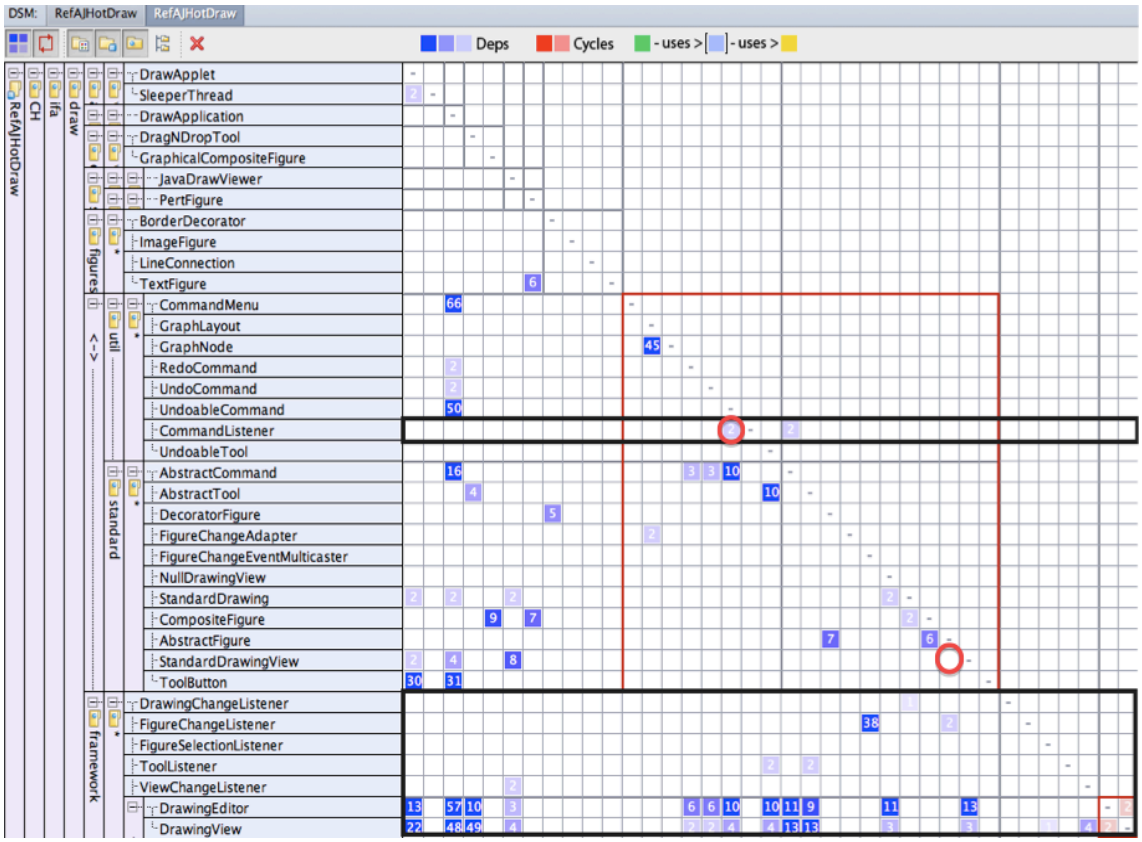

Figure 5.2: DSM - AJHotDraw

5.3.2 Prevayler

For the Prevayler, we found few instances of the observer pattern(three instances of the observer update shape and four instances from the pushing shape). Figure 5.3 shows the DSM for the classes where we found the pattern instances before and after the refactoring. Before aspect-refactoring modularization of the pattern instances, we see a tight coupling between the `the CentralPublisher`, `AbstractPublisher`, `POBox`, `PrevalentSystemGuard`, `TransactionTimestamp` and `TransactionSubscriber`. The first four types depend on the last two. A great deal of this dependency is caused by these types having to implement the observation functionality. For the classes `AbstractPublisher` and `PrevalentSystemGuard`, this coupling disappeared completely while it has reduced by one for `POBox`. Overall, aspect modularization of the observer pattern loosens the coupling between the types implementing the pattern.

Loose coupling is a sign of a well-designed and a well-structured software system. Loose coupling makes system maintenance much easier even over the evolved versions of the system.

The DSM of the entire Prevayler project can be found in appendix B.

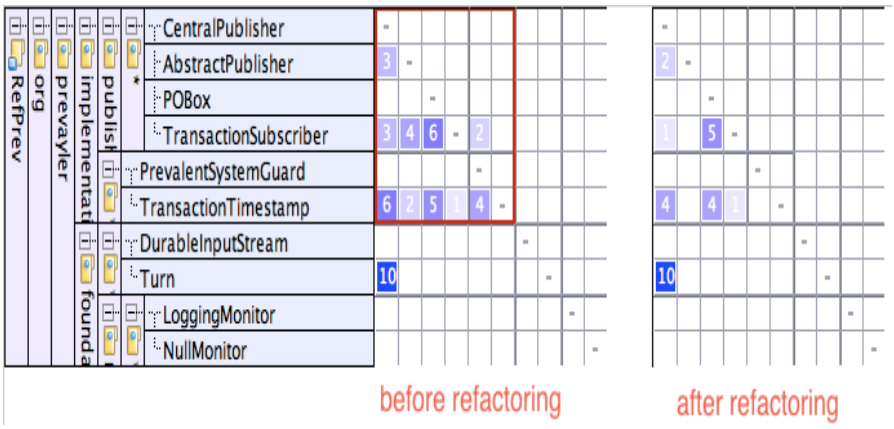

Figure 5.3: DSM of Prevayler

5.4 Time Assessment

As we mentioned in chapter 4, AJRefactor handles two kinds of observer pattern. Here we discuss the time required to refactor these two variations manually and with AJRefactor. We have used a wall-clock to track this time, beginning from the moment we see a pattern instance. For the manual refactoring most of the time was thinking versus typing. In both cases, the programmer needs to be aware of the entities involved in the refactoring. This includes the method that changes the subject state. Also, the different observers so he can correctly select them when AJRefactor pops up the wizard. When selecting wrong observers, AJRefactor is able to detect that this type is not eligible observer. Therefore, AJRefactor won't perform any changes to that type and no aspect will be created for it.

5.4.1 Observer Pattern Implemented with Pushing Technique

Figure 5.4 shows the time taken when refactoring the pattern instances implemented following the pushing technique to aspects. These instances encompass all of the occurrences, not just a subset. We measured the time in minutes. Each type in the Y-axis represents a completely observer instance. This means the subject and all of its observers and the different methods that changes the subject state and notifies the observers. As shown in the chart, AJRefactor could not refactor

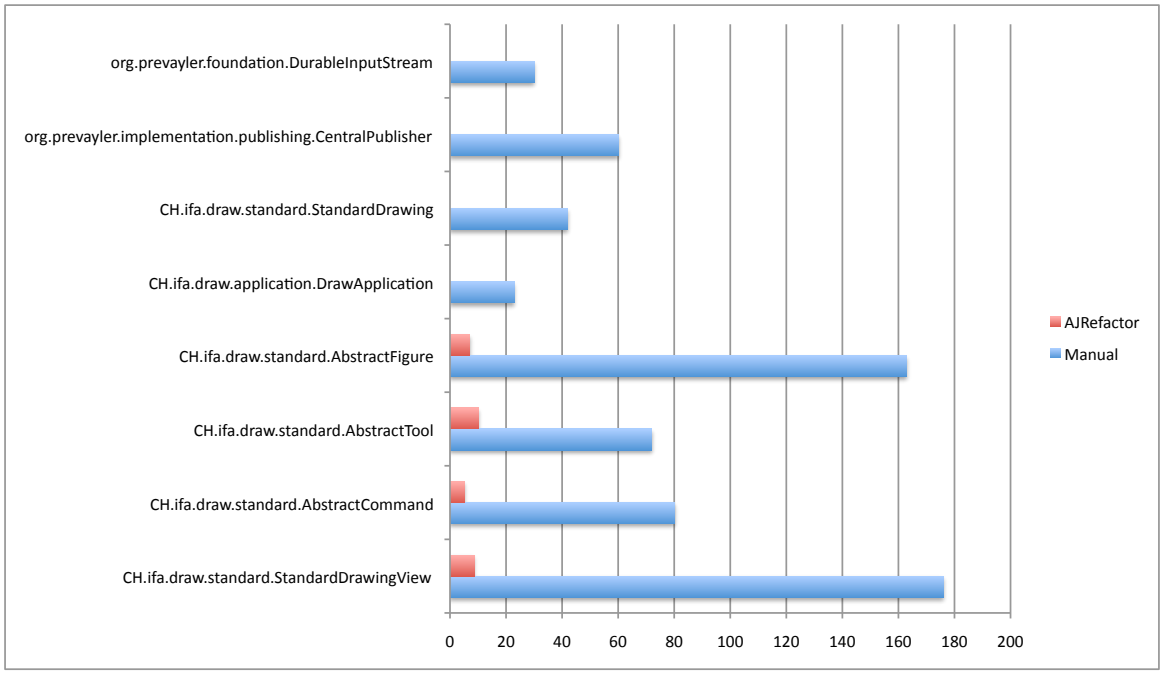

Figure 5.4: Refactoring Time - Pushing Technique

some of them for reasons we discussed in section 5.1.

The time consumed when manually refactoring observer pattern to aspects depends on the instance and the number of observers. Two-thirds of this time was dedicated to thinking wheres the other third was dedicated to typing. For example, when we refactored `CH.ifa.draw.application.DrawApplication` it took around 23 minutes as this type has only two observers. These two observe two different methods. The largest instances were `CH.ifa.draw.standard.AbstractFigure` with nine different observers and `CH.ifa.draw.standard.StandardDrawingView` with six different observers. It took 176 (2.9 hours) and 163 minutes (2.7 hours) respectively to manually refactor them while it took around 8 and 7 minutes respectively to automatically refactor them with AJRefactor. The time it takes to refactor these instances using AJRefactor also involves a human thinking. The reported time is mostly thinking versus this taken by the AJRefactor computations.

Comparing the manual refactoring experiment to the one performed with AJRefactor, we noticed that the time it takes to automatically refactor an instance represents on average 7% of the total time it takes to manually refactor the same instance. This percentage is very small which suggests there is a big time saving when refactoring with AJRefactor.

5.4.2 Refactoring Update Calls

Figure 5.5, shows the time in seconds for the update calls found in the different methods represented by the Y axis. When analyzing this time we found that on average it took 62 seconds to refactor observers' update calls with AJRefactor while on average it took three and a half minutes (215 seconds) to manually refactor these calls.

If we look at the total time it takes to refactor these instances as a whole, it takes 29 minutes automatically and 98 minutes manually. We see the automatic time is nearly the third of the time when manually refactoring these instances. Hence, cutting the refactoring time by the third. This considerably saves the programmer a good amount of time.

These results are for the instances that we were able to refactor with AJRefactor. The total time it took to manually refactor the ones that AJRefactor was not able to automatically refactor is 68 minutes. Manually refactoring these instances takes on average the same time it takes to manually refactor those that AJRefactor was able to refactor. This time is around four minutes. Since these instances still have to be manually refactored as they are not covered by AJRefactor, it is reasonable to compare the total refactoring time for all instances found when AJRefactor was used and when not used. That is a total of 166 minutes for a complete manual refactoring and 97 minutes for partially manual and partially semi-automatic. Again the time shows that we were able to cut the refactoring time by the third.

5.5 Reflection and Refactoring

Java reflection allows an executing Java program to examine and modify itself during the runtime. By first obtaining the `java.lang.Class` object, information about the class super type, the different fields and methods it has can be obtained. Using reflection might hinder the refactored code correctness. Part of refactoring the observer pattern involves deleting some fields and methods from the subject and observer classes. For example, observers field, the method that informs the observer of a change in the subject state, methods that manipulate observers, and the method that observers use to update their state. Although we delete these different elements from the subject and observer classes after ensuring they are not referenced elsewhere in the program, there are situations where deleting these entities might break the code correctness after refactoring. AJRefactor uses Eclipse searching engine to find references to any of these methods but unfortunately it cannot detect those used in reflection. Listing 5.2 shows an example. The code tries to get the method `notifyObservers()` from the class `MySubject`. Searching for references to this method does not include those used in reflection. Hence, deleting the method breaks code correctness. This special situation requires programmer awareness when refactoring a pattern in a program that uses reflection to make sure the program behaves the same after refactoring.

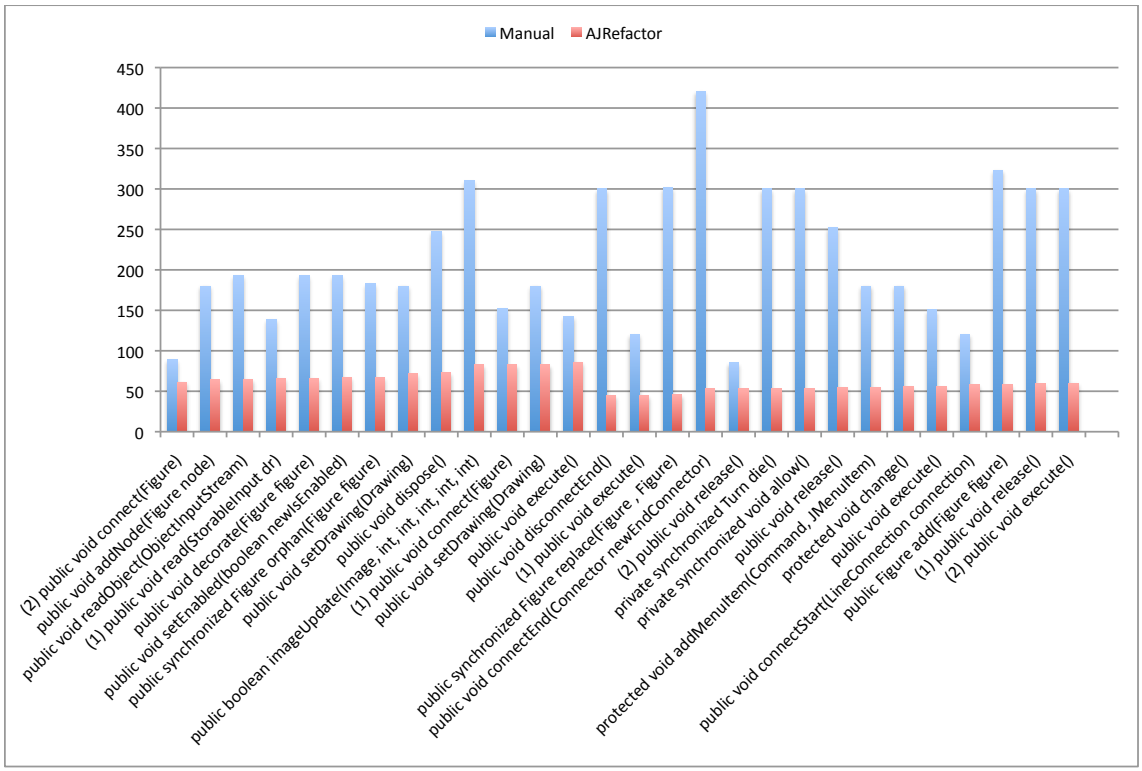

Figure 5.5: Refactoring Time - Observers Update Methods

```
try {
   Class clazz =  MySubject.class;
   Method updateMethod =  clazz.getClass().getMethod("notifyObservers", null);
} catch (Exception ex) {
  // handle exception case
}
```

Listing 5.2: Referencing Subject Methods through Reflection

5.6 Refactoring and Program Correctness

Based on our experience with both manual and semi-automatic refactoring of JHotDraw and Prevayler, we found that manual refactoring not only slower but also more error prone. Hence, automatic refactoring is more reliable. Although automatic refactoring is not 100% error free, one can anticipate the situations where program correctness might be at risk. AJRefactor helps with this by providing the following features

- *Previewing the change* AJRefactor features a change preview option for the classes effected by the refactoring before the refactoring can take place. Previewing these changes helps the programmer decide whether to accept these changes or just cancel the whole refactoring if there is a risk of breaking the program correctness and changing its behaviour.

- *Undoing the change* The programmer has previewed the changes and ensured the program behaves the same. After accepting the changes and completing the refactoring, there were errors that could not be anticipated before the refactoring. The simplest fix in this situation is to undo the whole refactoring and reverse the changes.

This is not the case for manual refactorings as errors can be easily introduced hindering program correctness with no chance to preview the changes before applying them or to undo them correctly after being applied.

5.7 AspectJ Performance Overhead

AspectJ community believes that code generated by AspectJ has insignificant performance overhead. Dufour et.al. [13] performed a study on the dynamic behaviour of eight different AspectJ applications. The performance of four of these projects confirmed the AspectJ community belief while three of them showed extremely high overhead, though the performance was comparing the base program (original Java program) to its equivalent aspect program. Depending on what and how AspectJ constructs being used the performance might get affected. The following gives some insights into the constructs that might impose an extra overhead.

- Inter-type declarations impose a very little overhead unless used to introduce new constructors.
- The `cflow` and `cflowbelow` constructs also increase the overhead. This is because AspectJ compiler (ajc) internal implementation of these constructs uses a stack data structure to track the matched join points. Bearing in mind the stack implementation has no equivalent in the base program.
- An `around` advice when it applies to itself. AspectJ has two different implementations to handle the call to `proceed()` either by inlining the code of the matched method or by using a closure. Closure is a function with an environment that allows this function to access all local variables declared in the scope where it was created. Closures are very expensive. Declaring an around advice that applies to itself enforces ajc to use closures strategy instead of inlining strategy which is more effective.
- Pointcuts that are very generic also impose a high performance overhead. Limiting the scope of the pointcut to match specific join points reduces the overhead.
- ajc tries to match join points based on the static context as possible but for some pointcuts such as `if` a runtime test still need to be performed which degrades the performance.

AJRefactor uses inter-type declarations to introduce methods and fields but not constructors. These aspects only employ after and before advice. The pointcut that our tool uses to match join points are `call, execution, if, target, and withincode`. Except for the `if` pointcut these pointcuts impose very little or no overhead. Hence, this leads us to believe there a little overhead imposed when introducing aspects to JHotDraw and Prevayler projects using our tool.

5.8 Multithreading and Refactoring

Java allows an application to handle two events at the same time. Each event has its own execution path. This is known as multithreading. There are two ways to implement threads in Java:

- by extending the `java.lang.Thread` class, or
- by implementing the `java.lang.Runnable` interface

Each thread needs to be started for it to run. This means calling the `start()` method on the thread object. The logic that the thread runs when started resides in the `public void run()` method that the thread should implement regardless of the way followed to implement the thread. The class `MyThread` in listing 5.3 implements the `Runnable` interface and implements the `run` method. This method prints the message associated with the `MyThread` object. In the `main`

```
class MyThread implements Runnable {
    private String msg;
    private Thread t;

    public MyThread(String s) {
        msg = s;
        t = new Thread(this);
        t.start();
    }

    public Thread getThread() { return t; }

    public void run() { System.out.println(msg); }

    public void synchronized synchronizedMethod(){...}

    public static void main(String args[]) {
            new MyThread("Hello");
            new MyThread("Synchronized");
            new MyThread("World");
    }
}
```

Listing 5.3: Thread Example

method three instances of the `MyThread` are created. Each instance has its own thread which get started after being created. Starting each thread means executing the `run` method which inturn prints the message associated with the currently executing `MyThread` object.

There are cases when two or more threads require access to a shared resource. Java provides a mechanism to synchronize the access to the shared resource. In non-synchronized multithreaded applications multiple threads might modify the shared resource at the same time which leaves the data in an inconsistent state. A thread needs to acquire the lock on the object associated with the shared resource. If more than one object needs to acquire the object lock, the system makes sure that only one thread acquire the object lock at a time. The thread holds the object lock till it completes the execution either normally or by throwing uncaught exception. There are two ways in which a synchronization can be implemented.

- executing a synchronized instance method of an object. Method `synchronizedMethod` in listing 5.3 shows an example.
- executing the body of a synchronized block on which an object synchronizes. The method `subscribe` in figure 5.6 shows an example

One question might arise in this situation is whether the refactoring would affect these running threads. Will they still run as intended? For all kinds of refactorings, a rule of thumb is to maintain program's correctness. Refactoring with AJRefactor is no exception. When the advised code is supposed to run as part of a synchronized method or block, the advice code will still only execute when the program reaches these join points defined in the pointcut. In case of a synchronized method, the refactoring will maintain the synchronized keyword; the pointcut and the advice will

be created as any other non-synchronized method. Also, in the case of the synchronized block, the refactoring will maintain the synchronized block and only the action that the advice executes when the program reaches its join point is moved into the advice part of the aspect. This means aspects maintains the execution path of any code resides in a synchronized method or a synchronized block. Figure 5.6 shows an example. The method `subscribe` updates the `_journal` as part of a synchronized block accessed by multiple threads. Now, when we advise this invocation with an after advice, this invocation will still be executed as part of the synchronized block when the `subscribePointcut` reaches its join point. The synchronized block remains untouched even after the refactoring. Overall, aspect-refactoring of an observer pattern that is part of a separate thread does not break the way these processes are supposed to be executed.

As we mentioned in chapter 4, AJRefactor currently does not recognize neither refactors situations where the subject change and the invocation of the method that notifies the observers are contained within a synchronization block.

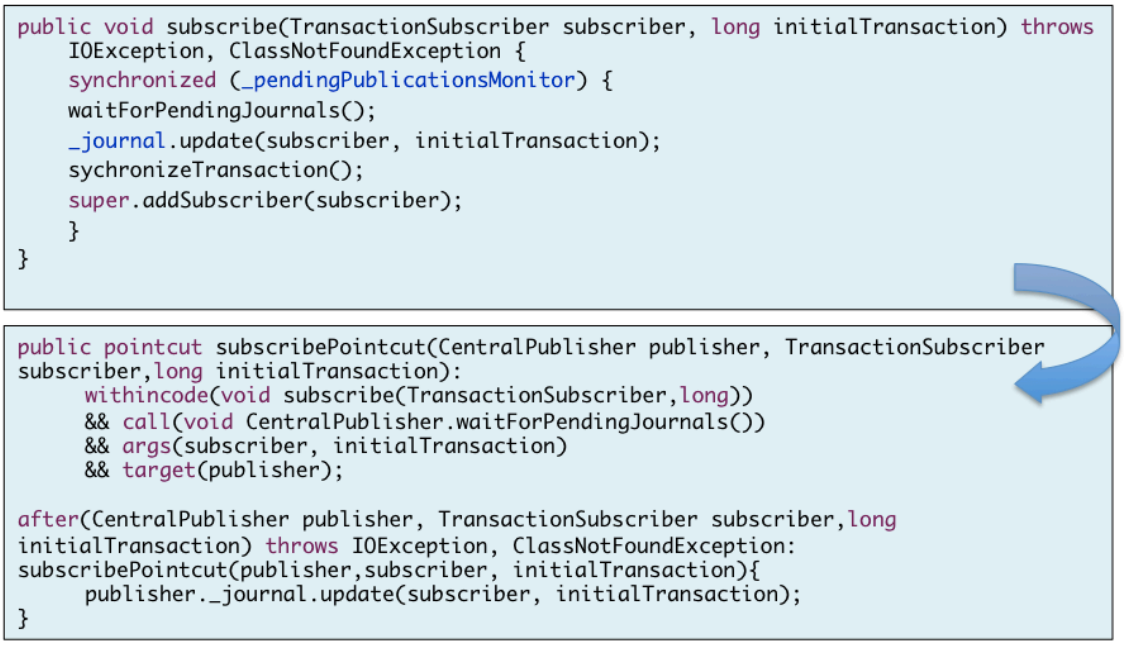

Figure 5.6: Example of Synchronized block

5.9 Volatile Variables and Refactoring

Java volatile keyword can only be used with variables. Declaring a variable as volatile tells threads not cache the variable and access its value from the main memory instead. For non-volatile variables, Java caches a copy of the variable in a register to enhance performance. If multiple threads modify this variable at the same time, each would see its own copy of the variable. One of the solutions

```
public class DelayWrite implements Runnable {
    private volatile String str;

    void setStr(String str) {
        this.str = str;
    }

    public void run() {
        System.out.println("Thread " + Thread.currentThread().getName() + " is running");
        while (str == null)
            ;
        System.out.println(str);
        System.out.println("Thread " + Thread.currentThread().getName() + " terminated");
    }

    public static void main(String[] args) {
        DelayWrite delay = new DelayWrite();
        new Thread(delay, "A").start();
        try {
            Thread.sleep(1000);
            new Thread(delay,"B").start();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        delay.setStr("Hello world!!");
    }
}
```

Listing 5.4: Volatile Variable in Multithreaded Program

to this issue would be to define the variable as volatile ensuring the most concurrent value of the volatile variable is visible to other threads. Listing 5.4 illustrates the use of volatile keyword in multi threaded program. Without declaring the `String` variable `str` as volatile, threads `A` and `B` will run forever because they won't be able to see the change in the value of `str`. Using volatile guarantees that the two threads will terminate after the `str` value being set to the string `"Hello world"`.

In the context of the observer pattern it is possible for this pattern to be used in multi threaded application. This means the observers list might be declared as volatile. This represents no issue when refactoring the pattern using AJRefactor. This is because the refactoring maintains variable declaration including its access modifiers. The filed `fSelectionListeners` in listing 5.5 is declared in the subject as volatile. When refactoring the pattern the field is moved into the aspect where its declaration as volatile variable maintained.

```
public class StandardDrawingView{
private  volatile Vector<FigureSelectionListener>  fSelectionListeners =
                                        new Vector<FigureSelectionListener>();
    ... }
public aspect StandardDrawingViewProtocol{
private  volatile Vector<FigureSelectionListener>  StandardDrawingView.
    fSelectionListeners = new Vector<FigureSelectionListener>();
    ...  }
```

Listing 5.5: Declaring Observers List as Volatile

```
public @interface MyAnnotation{
    String doSomething();
}

@MyAnnotation ("What to do")
public void foo() { .... }
```

Listing 5.6: User Defined Annotation Declaration

5.10 Java Annotations

Annotations provides information about the program but it does not affect the semantic of the program or how it suppose to run. Annotations has many uses, some of which include providing information to the compiler. These information helps the compiler detect errors or suppress warnings. Annotations can be used to annotate the declarations of program elements. For example, types , methods, fields, packages, local variables. The annotation declaration start with an sign followed by `interface` keyword followed by the annotation type name and a pair of brackets. The body of an annotation types can be empty, have a single element, or have multiple elements. An element of type `MethodDeclaration` cannot have parameters or throw declarations. The method return type must be one of the following types.

- primitive types (e.g. int, double, char)
- enum
- String
- an array of any of the previous types

The code listing below shows the declaration of a `MyAnnotation` annotation type and how we can use to annotate a `foo` method declaration. Since the annotation is declared with a single element, we can omit the element name or use the `value` keyword instead. The annotation of the `foo` method takes a single value which is the string "What to do".

There are mainly two types of annotation:

- Simple type annotations. These annotations are provided by Java Development Tool Kit (JDK) and are of three types.
  - Override: using this annotation type tells the compiler that this method should override a method from a super class. The compiler will detect an error if the method name was misspelled.
  - Deprecated: annotating a method with this annotation type helps the compiler detect an error if the deprecated method was reference else where in the program.

- Suppresswarnings: tells the compiler to suppress a warning

- Annotation of annotations. These are used to annotate an annotation type and are of four types.

  - Target: states the element type that the annotation applies on. For example,

    * `@Target(ElementType.TYPE)` annotates a type and its laments
    * `@Target(ElementType.FIELD)` annotates a field.
    * Other targets are method, parameter, constructor, local variable and an annotation type.

  - Retention: this annotation is used to state for how long and where an annotation will be retained which can be one of three types. Theses are

    * `RetentionPolicy.SOURCE` states that the annotation is retained at the source level and is ignored by the compiler.
    * `RetentionPolicy.CLASS` states that the annotation is retained by the compiler but is ignored by the Java virtual machine.
    * `RetentionPolicy.RUNTIME` states that the annotation is retained by the Java virtual machine at runtime.

  - Documented: states that annotations should be documented by javadoc tool which produces HTML documents for types.

  - Inherited: types annotated with this annotation means that subtypes inherits all the annotations from their super type.

When Java source code is compiled, annotations are processed by compilers plug-ins known as annotation processors. For example, Java 5 compiler has an annotation processing tool called APT. This tools produces additional Java source files and .class files. Java source files can be also compiled by Java compiler (javac). The apt tools is standard part of Java 6. This means no need for it to be used separately.

Since the annotation information are only available at the class files after the source code get compiled, it is possible for the observer pattern components to be part of an annotation declaration. For example, It is possible to use annotation to declare the the method that notifies the observer of subject state change. AJRefactor analyses the AST of the source code. In case of annotations, these information is not available to AJRefactor. Hence, pattern components declared using annotations are not applicable to our tool

5.11 Summary

Refactoring observer pattern instances to aspects improves programs modularity making them more focus on their original task. This modularity brings a few other enhances to the code. These includes

- reducing code size (LOC)
- reducing the total number of methods and classes
- loosening the coupling and decreasing the dependency between types involved in the pattern

AJRefactor provides an automatic support to refactor the observer pattern. Refactoring two Java applications JHotDraw and Prevayler with AJRefactor not only showed all the aforementioned benefits, but also speeds up the refactoring process. We also discuss the some of the features and technologies provided by and whether our tool to handle them in special way. Also, we discussed whether refactoring hinders the correctness of the program in general an in special situations such as when the pattern is implemented in a multithreaded application or when the observer or observers list is declared as volatile.

Chapter 6

Summary

This chapter summarizes our work relating to providing a tool support to automatically refactor the observer pattern to aspects. We begin by giving a brief overview of the problem and our solution to it. Then, we point out the contribution of our work along with the results of applying our tool on two Java applications: JHotDraw and Prevayler. Finally, we explain our future work.

6.1 Summary

Java code can improve modularization using aspects for situations where existing modularization techniques is not sufficient. Some of these situations include the implementation of design patterns. Implementing design patterns in Java makes programs more complex as the participating classes have to play more than one role at a time. This makes programs less readable and more difficult to maintain. Refactoring Java programs to aspects improves their modularity. AspectJ constructs provide a new opportunity to break programs down into modules. This is specially true when refactoring design patterns to aspects. Therefore, we contribute an Eclipse plug-in that semi-automates the refactoring of the subject pattern from Java to AspectJ. Refactoring the observer pattern into aspects allows the pattern participants to be free of any pattern code, easy to read and easy to maintain. Also, it permits the pattern participants to play more than one role at a time. Furthermore, it allows the participating classes to take part in multiple patterns.

Our plug-in AJRefactor refactors two different shapes of observer pattern: one where the pattern implementation follows the pushing technique and the other is where the observer calls its update method after the subject changes its state. For each case we handle three situations based on the location of subject notify call or observers update calls i.e. based on the parenting node. These situations are

- subject change and the call to subject notify or observer's update method are directly parented by the method declaration that hosts them, or
- subject change is the expression of an if statement while the call to subject notify or observer's update is in the then statement of this if, or

- subject change and the call to subject notify or observer's update method are both parented by a then statement of an if.

AJRefactor uses the Eclipse Java searching engine to find observer pattern elements. Unfortunately, this engine does not recognize AspectJ constructs. This means our tool applicability is limited to Java classes. Hence, it currently cannot be applied to AspectJ projects; i.e. it cannot perform AspectJ to AspectJ refactorings until AspectJ.org[1] extends the search engine. This is a place for AJRefactor improvement.

6.2 Contribution

The intended contribution of this thesis is to determine the value of refactoring Java code into aspects for appropriate situations where Java modularization methods are not sufficient. Our contribution to show these benefits includes the following.

- An Eclipse plug-in, AJRefactor, to automatically refactor the observer pattern using case studies. AJRefactor refactors mainly two shapes of the pattern. One that is implemented with the pushing technique and the other one is when the observer calls its update method after a subject changes its state.

- A measurement of the total instances AJRefactor was able to refactor. AJRefactor was able to refactor 85% of the pattern cases implemented with the pushing technique found in JHotDraw and Prevayler Java applications. In the other hand, AJRefactor was able to refactor 62% of the pattern cases where observers' call their update methods.

- A complete manual and semi-automatic refactorings of two Java applications with a measurement of the time it took in both cases. These applications vary in their size; One small and the other is medium. On average the time it takes to refactor observer instances with AJRefactor is 7% of the total time it takes to refactor the same instances manually. This shows there is a big time savings when automatically refactoring a pattern instance.

- A measurement of the Dependency Structure Matrix for each project before and after the refactoring. After refactoring to aspects, the dependency between types that are part of the observer pattern was loosened if not completely eliminated.

- A measurement of the number of lines of code before and after the refactoring. There was a reduction in the number of lines of code by 1.27% after refactoring JHotDraw and by 1.8% after refactoring Prevayler. Also, the number of methods has reduced by 5% in JHotDraw while the number of methods has remained the same for Prevayler. Finally, the number of types has reduced by 3% in JHotDraw while this number remained unchanged in Prevayler.

6.3 Future Work

In its current state, AJRefactor refactors only one pattern which is the observer pattern. Although, we tried to cover many common shapes that an observer instance could take, we found there is still some shapes that AJRefactor need to cover. Below, we outline our future work.

1. Adding more coverage to the cases that AJRefactor can refactor. Currently, AJRefactor refactors notify call or observer update calls where the parent node either a method declaration or an if statement. We would like to cover the cases below.
    - when the update call is contained within a try/catch block, or
    - when the notify call or an observer update call is contained within a synchronization block, or
    - when the observer update call is made from a class constructor, or
    - when the update call is parented by any of the loop blocks i.e. while, for, do-while, or enhanced for loop.
2. Modularity issues still persist when implementing other design patterns such as iterator, memento and strategy. Refactoring these patterns into aspects is one way to improve Java code modularity. Therefore, we would like to expand the tool functionality by adding the support to refactor more patterns.
3. Refactoring the calls to add and remove observer methods as part of the refactoring process.

References

[1] AspectJ Technology Project. http://www.eclipse.org/aspectj/.

[2] JHotDraw as Open-Source Project. http://www.jhotdraw.org/.

[3] Prevayler. http://prevayler.org/.

[4] CodePro AnalytiX. http://code.google.com/javadevtools/codepro /doc/features/metrics/metrics.html, Nov 2011.

[5] JastAddJRefactoring. http://jastadd.org, March 2011.

[6] AJHotDraw. http://ajhotdraw.sourceforge.net/, June 2012.

[7] IntelliJ IDEA. http://www.jetbrains.com/idea/, April 2012.

[8] Roberta Arcoverde, Patrícia Lustosa, Adeline Sousa, Sérgio Soares, and Paulo Borba. AJaTS AspectJ Transformation System: Tool Support for Aspect-Oriented Development and Refactoring. *XXI Brazilian Symposium on Software Engineering*, 2007.

[9] Dave Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Automated Refactoring of Object Oriented Code into Aspects. In *ICSM*, pages 27–36, 2005.

[10] David Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Tool-Supported Refactoring of Existing Object-Oriented Code into Aspect. *IEEE Software*, 32(9):698–717, September 2006.

[11] Paulo Borba and Sergio Soare. Refactoring and Code Generation Tools for AspectJ. In *Tools for Aspect-Oriented Software Development*, November 2002.

[12] Mariano Ceccato. *Migrating Object Oriented code to Aspect Oriented Programming.* PhD thesis, University of Trento, 2000.

[13] Bruno Dufour, Christopher Goard, Laurie Hendren, Oege De Moor, Ganesh Sittampalam, and Clark Verbrugge. Measuring the Dynamic Behaviour of AspectJ Programs. In *In Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 150–169, 2004.

[14] Eclipse. Eclipse Documentation. http://help.eclipse.org/indigo/index.jsp.

[15] Torbjörn Ekman, Max Schäfer, and Mathieu Verbaere. Refactoring is not (yet) about transformation. In *Proceedings of the 2nd Workshop on Refactoring Tools*, pages 1–5, 2008.

[16] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring Improving The Design Of Existing Code.* Addison Wesley, 1999.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[18] Irum Iqbal Godil. *An Open Infrastructure for Refactoring Aspects.* PhD thesis, University of Torno, 2006.

[19] William G. Griswold, Yoshikiyo Kato, and Jimmy J. Yuan. Aspect Browser: Tool Support for Managing Dispersed Aspects. In *First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems*, volume 99, pages 1–6, 1999.

[20] Jan Hannemann and Gregor Kiczales. Design Pattern Implementation in Java and AspectJ. In *OOPSLA*, volume 37, pages 161–173, November 2002.

[21] Raffi Khatchadourian, Phil Greenwood, Awais Rashid, and Guoqing Xu. Pointcut Rejuvenation: Recovering Pointcut Expressions in Evolving. In *ASE*, pages 575–579, 2009.

[22] Thomas Kuhn and Olivier Thomann. Abstract syntax tree. http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html, November 2006.

[23] Ramnivas Laddad. *AspectJ in Action*. Manning Publications, Greenwich, CT, 2003.

[24] Rajeev Motwani Lawrence Page, Sergey Brin and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford InfoLab, 1999.

[25] Marius Marin. Identifying Crosscutting Concerns Using Fan-in Analysis. *TOSEM*, 17(1):1–37, December 2007.

[26] Marius Marin, Leon Moonen, and Arie van Deursen. An Approach to Aspect Refactoring Based on Crosscutting Concern Types. In *MACS '05 Proceedings of the 2005 workshop on Modeling and analysis of concerns in software*, volume 30, pages 1–5, July 2005.

[27] Marius Marin, Leon Moonen, and Arie van Deursen. An Integrated Crosscutting Concern Migration Strategy and its Application to JHotDraw. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 101–110, 2007.

[28] Miguel Pessoa Monteiro. *Catalogue of Refactorings for AspectJ*. Technical report, Minho University, December 2004.

[29] Ricardo Nusca. AspectRefactor. http://sourceforge.net/projects/aspectrefactor, June 2006.

[30] Martin P. Robillard and Gail C. Murphy. Representing Concerns in Source Code. *TOSEM*, 16(1):1–38, 2007.

[31] Martin P. Robillard and Frederic WeigandWarr. ConcernMapper: Simple ViewBased Separation of Scattered Concerns. In *OOPSLA workshop on Eclipse technology eXchange*, pages 65–69, 2005.

[32] Shimon Rura and Barbara Lerner. A Basis for AspectJ Refactoring. 2004.

[33] Macneil Shonle, Jonathan Neddenriep, and William Griswold. AspectBrowser for Eclipse: A Case Study in Plug-in Retargeting. In *OOPSLA workshop on eclipse technology eXchange*, pages 78–82, 2004.

[34] Paolo Tonella and Mariano Ceccato. Aspect Mining through the Formal Concept Analysis of Execution Traces. In *11th Working Conference on Reverse Engineering*, pages 112–121, 2004.

[35] Jan Wloka. Aspect-aware Refactoring tool support. *Third ACM Workshop on Refactoring Tools*, October 2009.

[36] Charles Zhang and Hans-Arno Jacobsen. Efficiently Mining Crosscutting Concerns through RandomWalks. In *AOSD*, pages 226–238, 2007.

Appendix A

Observer Pattern Instances

This appendix shows the the observer pattern instances we have refactored in details along with those ones we could not refactor.

A.1 Observer Pattern Implemented with Pushing Technique

Table A.1 shows the details of each of the observer instances we found in both projects JHotDraw and Prevayler. Dark grey highlighted cells are this ones that AJRefactor was not able to refactor.

A.2 Refactoring Update Calls

Table A.2 shows the details of the observers' update calls found in both applications JHotDraw and Prevayler. Dark grey highlighted cells are this ones that AJRefactor was not able to refactor.

SR#	Subject	Observers	Subject Interface	Changing Method	Changing Method Shape	Observer Interface
1	CH.ifa.draw.standard.StandardDrawingView	CH.ifa.draw.standard.AbstractCommand	CH.ifa.draw.framework.DrawingView	public void clearSelection()	MethodDecSubjectChange - NoifyCall	CH.ifa.draw.framework.FigureSelectionListener
2	CH.ifa.draw.standard.StandardDrawingView	CH.ifa.draw.samples.javadraw.JavaDrawViewer	CH.ifa.draw.framework.DrawingView	public void clearSelection()	MethodDecSubjectChange - NoifyCall	CH.ifa.draw.framework.DrawingEditor
3	CH.ifa.draw.standard.StandardDrawingView	CH.ifa.draw.applet.DrawApplet	CH.ifa.draw.framework.DrawingView	public void clearSelection()	MethodDecSubjectChange - NoifyCall	CH.ifa.draw.framework.DrawingEditor
4	CH.ifa.draw.standard.StandardDrawingView	CH.ifa.draw.application.DrawApplication	CH.ifa.draw.framework.DrawingView	public void clearSelection()	MethodDecSubjectChange - NoifyCall	CH.ifa.draw.framework.DrawingEditor
5	CH.ifa.draw.standard.StandardDrawingView	CH.ifa.draw.util.UndoableCommand	CH.ifa.draw.framework.DrawingView	public void clearSelection()	MethodDecSubjectChange - NoifyCall	CH.ifa.draw.framework.FigureSelectionListener
6	CH.ifa.draw.standard.StandardDrawingView	CH.ifa.draw.standard.AbstractCommand	CH.ifa.draw.framework.DrawingView	public void toggleSelection(Figure figure)	MethodDecSubjectChange - NoifyCall	CH.ifa.draw.framework.FigureSelectionListener
7	CH.ifa.draw.standard.StandardDrawingView	CH.ifa.draw.util.UndoableCommand	CH.ifa.draw.framework.DrawingView	public void toggleSelection(Figure figure)	MethodDecSubjectChange - NoifyCall	CH.ifa.draw.framework.FigureSelectionListener
8	CH.ifa.draw.standard.StandardDrawingView	CH.ifa.draw.samples.javadraw.JavaDrawViewer	CH.ifa.draw.framework.DrawingView	public void toggleSelection(Figure figure)	MethodDecSubjectChange - NoifyCall	CH.ifa.draw.framework.DrawingEditor
9	CH.ifa.draw.standard.StandardDrawingView	CH.ifa.draw.applet.DrawApplet	CH.ifa.draw.framework.DrawingView	public void toggleSelection(Figure figure)	MethodDecSubjectChange - NoifyCall	CH.ifa.draw.framework.DrawingEditor
10	CH.ifa.draw.standard.StandardDrawingView	CH.ifa.draw.application.DrawApplication	CH.ifa.draw.framework.DrawingView	public void toggleSelection(Figure figure)	MethodDecSubjectChange - NoifyCall	CH.ifa.draw.framework.DrawingEditor
11	CH.ifa.draw.standard.StandardDrawingView	CH.ifa.draw.standard.AbstractCommand	CH.ifa.draw.framework.DrawingView	public void addToSelection(Figure figure)	ThenStmntSubjectChange - NotifyCall	CH.ifa.draw.framework.FigureSelectionListener
12	CH.ifa.draw.standard.StandardDrawingView	CH.ifa.draw.samples.javadraw.JavaDrawViewer	CH.ifa.draw.framework.DrawingView	public void addToSelection(Figure figure)	ThenStmntSubjectChange - NotifyCall	CH.ifa.draw.framework.DrawingEditor
13	CH.ifa.draw.standard.StandardDrawingView	CH.ifa.draw.applet.DrawApplet	CH.ifa.draw.framework.DrawingView	public void addToSelection(Figure figure)	ThenStmntSubjectChange - NotifyCall	CH.ifa.draw.framework.DrawingEditor
14	CH.ifa.draw.standard.StandardDrawingView	CH.ifa.draw.application.DrawApplication	CH.ifa.draw.framework.DrawingView	public void addToSelection(Figure figure)	ThenStmntSubjectChange - NotifyCall	CH.ifa.draw.framework.DrawingEditor
15	CH.ifa.draw.standard.StandardDrawingView	CH.ifa.draw.util.UndoableCommand	CH.ifa.draw.framework.DrawingView	public void addToSelection(Figure figure)	ThenStmntSubjectChange - NotifyCall	CH.ifa.draw.framework.FigureSelectionListener
16	CH.ifa.draw.standard.StandardDrawingView	CH.ifa.draw.standard.AbstractCommand	CH.ifa.draw.framework.DrawingView	public void removeFromSelection(Figure)	ThenStmntSubjectChange - NotifyCall	CH.ifa.draw.framework.FigureSelectionListener
17	CH.ifa.draw.standard.StandardDrawingView	CH.ifa.draw.util.UndoableCommand	CH.ifa.draw.framework.DrawingView	public void removeFromSelection(Figure)	ThenStmntSubjectChange - NotifyCall	CH.ifa.draw.framework.FigureSelectionListener
18	CH.ifa.draw.standard.StandardDrawingView	CH.ifa.draw.applet.DrawApplet	CH.ifa.draw.framework.DrawingView	public void removeFromSelection(Figure)	ThenStmntSubjectChange - NotifyCall	CH.ifa.draw.framework.DrawingEditor
19	CH.ifa.draw.standard.StandardDrawingView	CH.ifa.draw.application.DrawApplication	CH.ifa.draw.framework.DrawingView	public void removeFromSelection(Figure)	ThenStmntSubjectChange - NotifyCall	CH.ifa.draw.framework.DrawingEditor
20	CH.ifa.draw.standard.StandardDrawingView	CH.ifa.draw.util.UndoableCommand	CH.ifa.draw.framework.DrawingView	public void removeFromSelection(Figure)	ThenStmntSubjectChange - NotifyCall	CH.ifa.draw.framework.FigureSelectionListener
21	CH.ifa.draw.standard.AbstractCommand	CH.ifa.draw.util.CommandMenu	CH.ifa.draw.util.CommandListener	viewSelectionChanged(DrawingView, Drawi	IfExpSubjectChange – NotifyCall	CH.ifa.draw.util.CommandListener
22	CH.ifa.draw.standard.AbstractCommand	CH.ifa.draw.util.UndoableCommand	CH.ifa.draw.util.CommandListener	viewSelectionChanged(DrawingView, Drawi	IfExpSubjectChange – NotifyCall	CH.ifa.draw.util.CommandListener
23	CH.ifa.draw.standard.AbstractCommand	CH.ifa.draw.util.CommandMenu	CH.ifa.draw.util.CommandListener	viewSelectionChanged(DrawingView, Drawi	IfExpSubjectChange – NotifyCall	CH.ifa.draw.util.CommandListener
24	CH.ifa.draw.standard.AbstractCommand	CH.ifa.draw.util.UndoableCommand	CH.ifa.draw.util.CommandListener	viewSelectionChanged(DrawingView, Drawi	IfExpSubjectChange – NotifyCall	CH.ifa.draw.util.CommandListener
25	CH.ifa.draw.standard.AbstractTool	CH.ifa.draw.standard.ToolButton	ITool	activate()	ThenStmntSubjectChange - NotifyCall	CH.ifa.draw.framework.ToolListener
26	CH.ifa.draw.standard.AbstractTool	CH.ifa.draw.standard.ToolButton	ITool	deactivate()	ThenStmntSubjectChange - NotifyCall	CH.ifa.draw.framework.ToolListener
27	CH.ifa.draw.standard.AbstractTool	CH.ifa.draw.standard.ToolButton	ITool	setUsable(boolean)	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.ToolListener
28	CH.ifa.draw.standard.AbstractTool	CH.ifa.draw.standard.ToolButton	ITool	setUsable(boolean)	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.ToolListener
29	CH.ifa.draw.standard.AbstractTool	CH.ifa.draw.standard.ToolButton	ITool	setEnabled(boolean)	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.ToolListener
30	CH.ifa.draw.standard.AbstractTool	CH.ifa.draw.util.UndoableTool	ITool	activate()	ThenStmntSubjectChange - NotifyCall	CH.ifa.draw.framework.ToolListener
31	CH.ifa.draw.standard.AbstractTool	CH.ifa.draw.util.UndoableTool	ITool	deactivate()	ThenStmntSubjectChange - NotifyCall	CH.ifa.draw.framework.ToolListener
32	CH.ifa.draw.standard.AbstractTool	CH.ifa.draw.util.UndoableTool	ITool	setUsable(boolean)	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.ToolListener
33	CH.ifa.draw.standard.AbstractTool	CH.ifa.draw.util.UndoableTool	ITool	setUsable(boolean)	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.ToolListener
34	CH.ifa.draw.standard.AbstractTool	CH.ifa.draw.util.UndoableTool	ITool	setEnabled(boolean)	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.ToolListener
35	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.figures.TextFigure	IFigure	release()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.FigureChangeListener
36	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.contrib.GraphicalCompositeFigure	IFigure	release()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.FigureChangeListener
37	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.standard.FigureChangeEventMultic	IFigure	release()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.FigureChangeListener
38	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.standard.FigureChangeAdapter	IFigure	release()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.FigureChangeListener
39	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.standard.DecoratorFigure	IFigure	release()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.FigureChangeListener
40	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.standard.CompositeFigure	IFigure	release()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.FigureChangeListener
41	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.samples.pert.PertFigure	IFigure	release()	IfExpSubjectChange – NotifyCall	
42	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.standard.StandardDrawing	IFigure	invalidate()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.Drawing
43	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.figures.LineConnection	IFigure	invalidate()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.ConnectionFigure
44	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.figures.TextFigure	IFigure	invalidate()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.FigureChangeListener
45	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.util.GraphLayout	IFigure	invalidate()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.FigureChangeListener
46	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.contrib.GraphicalCompositeFigure	IFigure	invalidate()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.FigureChangeListener
47	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.standard.FigureChangeEventMultic	IFigure	invalidate()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.FigureChangeListener
48	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.standard.FigureChangeAdapter	IFigure	invalidate()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.FigureChangeListener
49	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.standard.DecoratorFigure	IFigure	invalidate()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.FigureChangeListener
50	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.standard.CompositeFigure	IFigure	invalidate()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.FigureChangeListener
51	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.figures.BorderDecorator	IFigure	invalidate()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.FigureChangeListener
52	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.figures.LineConnection	IFigure	changed()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.ConnectionFigure
53	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.figures.TextFigure	IFigure	changed()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.FigureChangeListener
54	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.util.GraphLayout	IFigure	changed()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.FigureChangeListener
55	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.standard.FigureChangeEventMultic	IFigure	changed()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.FigureChangeListener
56	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.standard.FigureChangeAdapter	IFigure	changed()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.FigureChangeListener
57	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.standard.DecoratorFigure	IFigure	changed()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.FigureChangeListener
58	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.standard.CompositeFigure	IFigure	changed()	IfExpSubjectChange – NotifyCall	CH.ifa.draw.framework.FigureChangeListener
59	CH.ifa.draw.standard.AbstractFigure	CH.ifa.draw.samples.pert.PertFigure	IFigure	changed()	IfExpSubjectChange – NotifyCall	
60	CH.ifa.draw.application.DrawApplication	CH.ifa.draw.standard.AbstractCommand	IView	createDrawingView(Drawing)		CH.ifa.draw.framework.ViewChangeListener
61	CH.ifa.draw.application.DrawApplication	CH.ifa.draw.standard.AbstractTool	IView	createDrawingView(Drawing)		CH.ifa.draw.framework.ViewChangeListener
62	CH.ifa.draw.application.DrawApplication	CH.ifa.draw.standard.AbstractCommand	IView	setView(DrawingView)	ThenStmntSubjectChange - NotifyCall	CH.ifa.draw.framework.ViewChangeListener
63	CH.ifa.draw.application.DrawApplication	CH.ifa.draw.standard.AbstractTool	IView	setView(DrawingView)	ThenStmntSubjectChange - NotifyCall	CH.ifa.draw.framework.ViewChangeListener
64	CH.ifa.draw.application.DrawApplication	CH.ifa.draw.application.DrawApplet	IView			
65	CH.ifa.draw.standard.StandardDrawing	CH.ifa.draw.standard.StandardDrawingView	IDrawing	remove(Figure figure)		CH.ifa.draw.framework.DrawingChangeListener
66	CH.ifa.draw.standard.StandardDrawing	CH.ifa.draw.standard.NullDrawingView	IDrawing	remove(Figure figure)		CH.ifa.draw.framework.DrawingChangeListener
67	org.prevayler.implementation.publishing.Cer	org.prevayler.implementation.publishing.POBox	org.prevayler.implementation.publish	notifySubscribers(TransactionGuide)		
68	org.prevayler.implementation.publishing.Cer	org.prevayler.implementation.PrevalentSystem	org.prevayler.implementation.publish	notifySubscribers(TransactionGuide)		
69	org.prevayler.foundation.DurableInputStrea	org.prevayler.foundation.monitor.LoggingMonitor		subscribe(TransactionSubscriber, long)		
70	org.prevayler.foundation.DurableInputStrea	org.prevayler.foundation.monitor.NullMonitor		subscribe(TransactionSubscriber, long)		

Figure A.1: Details of refactoring pattern instances implemented with pushing technique to aspects

#	Subject	Update Calls	Declaring Method	Shape
1	CH.ifa.draw.util.UndoableCommand	getDrawingEditor().figureSelectionChanged(view());	public void execute()	ifExpSubjectChange - multiInvocations
2	CH.ifa.draw.standard.StandardDrawingView	fDrawing.removeDrawingChangeListener(this)	public void setDrawing(Drawing)	ifExpSubjectChange - multiInvocations
3	CH.ifa.draw.standard.StandardDrawingView	fDrawing.addDrawingChangeListener(this);	public void setDrawing(Drawing)	ifExpSubjectChange - multiInvocations
4	CH.ifa.draw.standard.AbstractCommand	view().removeFigureSelectionListener(this);	public void dispose()	ifExpSubjectChange - multiInvocations
5	CH.ifa.draw.figures.TextFigure	fObservedFigure.addFigureChangeListener(this);	public void read(StorableInput dr)	ifExpSubjectChange - multiInvocations
6	CH.ifa.draw.figures.TextFigure	fObservedFigure.addFigureChangeListener(this);	public void readObject(ObjectInputStream s)	ifExpSubjectChange - multiInvocations
7	CH.ifa.draw.figures.TextFigure	fObservedFigure.removeFigureChangeListener(this);	public voidconnect(Figure)	ifExpSubjectChange - multiInvocations
8	CH.ifa.draw.figures.LineConnection	startFigure().removeFigureChangeListener(this);	public void release()	ifExpSubjectChange - multiInvocations
9	CH.ifa.draw.figures.LineConnection	endFigure().removeFigureChangeListener(this);	public void release()	ifExpSubjectChange - multiInvocations
10	CH.ifa.draw.contrib.GraphicalCompositeFigure	listener().figureRequestUpdate(new FigureChangeEvent(this));	public void change()	ifExpSubjectChange - multiInvocations
11	CH.ifa.draw.figures.ImageFigure	listener().figureRequestUpdate(new FigureChangeEvent(this));	public void imageUpdate(Image, int, int, int, int, int)	ifExpSubjectChange - multiInvocations
12	CH.ifa.draw.standard.AbstractTool	getEventDispatcher().fireToolDisabledEvent();	public void setEnabled(boolean newIsEnabled)	ifExpSubjectChange - multiInvocations
13	CH.ifa.draw.standard.CompositeFigure	replacement.addToContainer(this); figure.removeFromContainer(this);	public synchronized Figure replace(Figure, Figure)	ifExpSubjectChange - multiInvocations
14	CH.ifa.draw.standard.CompositeFigure	figure.removeFromContainer(this);	public synchronized Figure orphan(Figure figure)	ifExpSubjectChange - multiInvocations
15	CH.ifa.draw.util.UndoableCommand	view().addFigureSelectionListener(this);	public void execute()	MethodDecSubjectChange - multiInvocations
16	CH.ifa.draw.util.RedoCommand	getDrawingEditor().figureSelectionChanged(lastRedoable.getDrawingView());	public void execute()	MethodDecSubjectChange - multiInvocations
17	CH.ifa.draw.util.GraphLayout	node.addFigureChangeListener(this);	public void addNode(Figure node)	MethodDecSubjectChange - multiInvocations
18	CH.ifa.draw.util.CommandMenu	command.addCommandListener(this);	protected void addMenuItem(Command, JMenuItem)	MethodDecSubjectChange - multiInvocations
19	CH.ifa.draw.standard.DecoratorFigure	getDecoratedFigure().removeFromContainer(this);	public void release()	MethodDecSubjectChange - multiInvocations
20	CH.ifa.draw.figures.TextFigure	fObservedFigure.addFigureChangeListener(this);	public void connect(Figure)	MethodDecSubjectChange - multiInvocations
21	CH.ifa.draw.figures.LineConnection	connection.startFigure().addFigureChangeListener(connection);	public void connectStart(LineConnection connection)	MethodDecSubjectChange - multiInvocations
22	CH.ifa.draw.figures.LineConnection	endFigure().addFigureChangeListener(this);	public void connectEnd(Connector newEndConnector)	MethodDecSubjectChange - multiInvocations
23	CH.ifa.draw.figures.LineConnection	endFigure().removeFigureChangeListener(this);	public void disconnectEnd()	MethodDecSubjectChange - multiInvocations
24	CH.ifa.draw.standard.DecoratorFigure	fComponent.addToContainer(this);	public void decorate(Figure figure)	MethodDecSubjectChange - multiInvocations
25	CH.ifa.draw.util.UndoCommand	getDrawingEditor().figureSelectionChanged(lastUndoable.getDrawingView())	public void execute()	MethodDecSubjectChange - multiInvocations
26	CH.ifa.draw.standard.CompositeFigure	figure.addToContainer(this);	public Figure add(Figure figure)	ThenStmntSubjectChange - multiInvocations
27	org.prevayler.foundation.Turn	notifyAll();	private synchronized Turn die()	MethodDecSubjectChange - multiInvocations
28	org.prevayler.foundation.Turn	notifyAll();	private synchronized void allow()	MethodDecSubjectChange - multiInvocations
29	CH.ifa.draw.standard.DecoratorFigure	getDecoratedFigure().addToContainer(this);	private void readObject(ObjectInputStream s)	
30	CH.ifa.draw.figures.TextFigure	disconnectFigure.removeFigureChangeListener(this);	public void disconnect(Figure disconnectFigure)	
31	CH.ifa.draw.util.UndoableTool	getWrappedTool().addToolListener(tool)	UndoableTool	
32	CH.ifa.draw.util.UndoableCommand	getWrappedCommand().addCommandListener(this)	UndoableCommand(Command)	

33	CH.ifa.draw.util.GraphLayout	if (nodes != null) {
		Enumeration nodeEnum = nodes.keys();
		while (nodeEnum.hasMoreElements()) {
		Figure node = (Figure) nodeEnum.nextElement();
		node.removeFigureChangeListener(this);
		}
		nodes = null;
		edges = null;
		}
			public void remove()	

34	CH.ifa.draw.standard.StandardDrawingView	fSelectionListeners = new Vector();	StandardDrawingView(DrawingEditor, int, int)	
35	CH.ifa.draw.standard.StandardDrawingView	addFigureSelectionListener(editor());	StandardDrawingView(DrawingEditor, int, int)	
36	CH.ifa.draw.standard.StandardDrawingView	fSelectionListeners = new Vector<FigureSelectionListener>()	public void readObject(ObjectInputStream s)	
37	CH.ifa.draw.standard.DecoratorFigure	getDecoratedFigure().removeFromContainer(this)	public Figure peelDecoration()	
38	CH.ifa.draw.figures.LineConnection	startFigure().removeFigureChangeListener(this);	public void disconnectStart()	
39	CH.ifa.draw.standard.StandardView	figure.listener().figureRequestRemove(new FigureChangeEvent(figure, null));	public synchronized Figure remove(Figure figure)	
40	org.prevayler.implementation.publishing.Centr	_journal.update(subscriber, initialTransaction);	subscribe(TransactionSubscriber, long)	
41	CH.ifa.draw.standard.AbstractFigure	removeFigureChangeListener(c);	public void removeFromContainer(FigureChangeListener c)	
42	CH.ifa.draw.standard.AbstractFigure	addFigureChangeListener(c);	public void addToContainer(FigureChangeListener c)	

43	CH.ifa.draw.standard.CompositeFigure	FigureEnumeration fe = figures();
		while (fe.hasMoreElements()) {
		Figure figure = fe.nextFigure();
		figure.addToContainer(this);
		}
			public void readObject(ObjectInputStream)	

44	CH.ifa.draw.standard.CompositeFigure	FigureEnumeration fe = figures();	public void removeAll()	
45	CH.ifa.draw.standard.AbstractCommand	getDrawingEditor().addViewChangeListener(this);	public AbstractCommand(String, DrawingEditor , boolean)	

Figure A.2: Details of refactoring observers' update calls to aspects

Appendix B

Dependency Structure Matrix

This appendix shows the DSM of entire Prevayler project before and after the refactoring.

B.1 DSM of Prevayler Before Refactoring

Figure B.1 shows the complete DSM of the Prevayler project before refactoring the observer pattern.

B.2 DSM of Prevayler After Refactoring

Figure B.1 shows the complete DSM of the Prevayler project after refactoring the observer pattern.

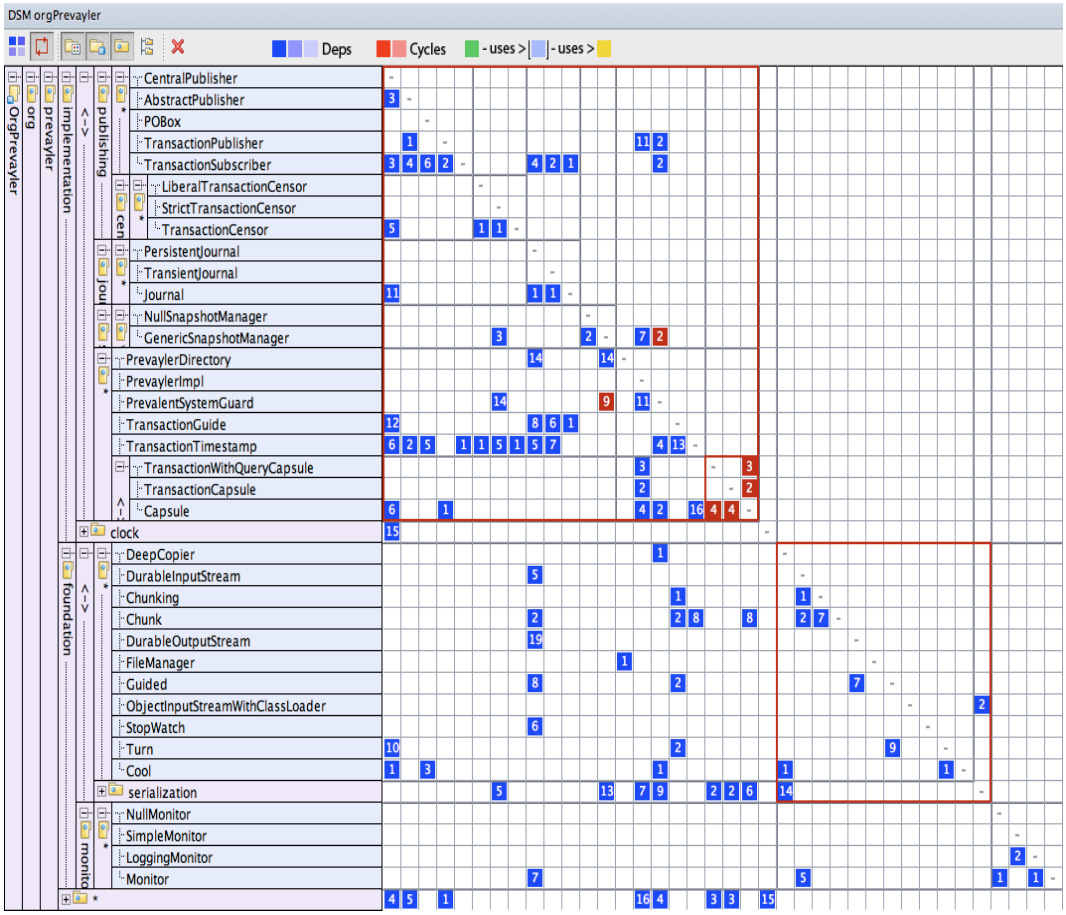

Figure B.1: DSM of Prevayler Before Refactoring

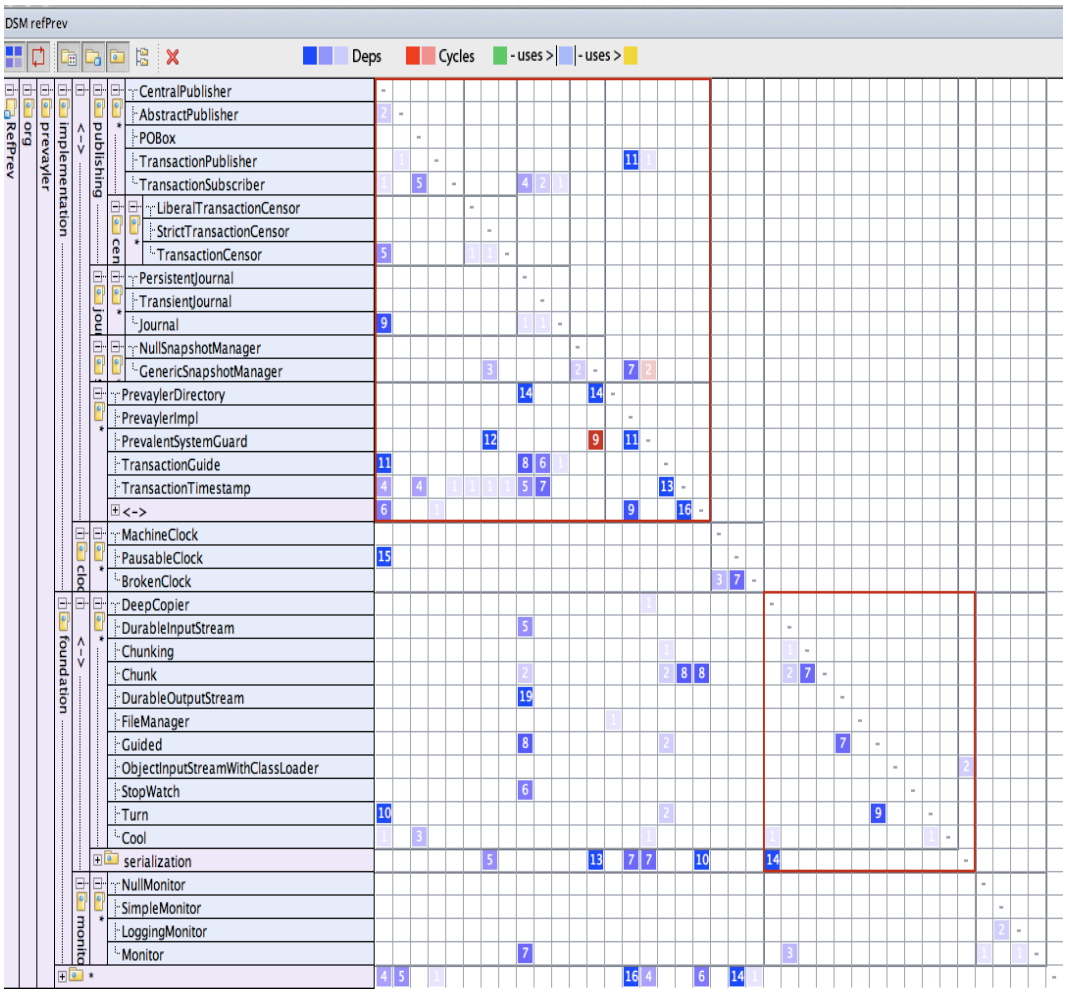

Figure B.2: DSM of Prevayler After Refactoring

Appendix C

AJRefactor Plugin Source Code

To get a copy of AJRefactor source code please contact my supervisor Professor Christopher Dutchyn at cjd032@cs.usask.ca